Herong's Notes on JSP

Version 3.03

Dr. Herong Yang

Copyright © 2001 - 2003 by Dr. Herong Yang. All rights reserved.

Table of Contents

About This Book

Tomcat

- Installing Tomcat 4.1.18
- Using Tomcat as a Web Server
- "Hello world!" Example

JavaServer Pages (JSP)

- What is JSP?
- "Hello world!" Java Servlet Class
- Writing JSP Servlet Class Directly

Execution Context

- Predefined Objects
- The "request" Object
- The "session" Object

JSP Elements

- Syntactic Elements of a JSP Page
- Writing JSP Pages in XML Format
- Directive Elements
- Action Elements

- JSP Example - "CurrentTime.jsp"

JSP Sessions and Debugging

- What Is a Session?
- The "session" Object
- Passing Values between Pages
- HTTP Communication Level Debugging

Using JavaBean Classes

- The "jsp:useBean" Action Elements
- Compilation Errors with Tomcat 4.1.18 and JDK 1.4.1
- Setting and Getting JavaBeans Properties
- Using JavaBeans as Objects in Scripting Elements
- Using Java Objects as JavaBeans
- Refreshing the Loaded Bean Classes

Using Cookies

- What is a Cookie?
- Sending and Receiving Cookies
- Persistent Cookies

Controlling HTTP Response Header Lines

- HTTP Response Syntax
- HTTP Response Header Lines
- Controlling Response Header Lines
- Viewing Response Header Lines
- Response Header Lines of Static Files
- Response Header Lines Affected by jsp:directive.page Elements
- Setting Header Lines Directly in JSP Pages
- Generating Non-HTML Entity Body
- IE 6.0 Bug on Display PDF Data

Localization / Internationalization - Non ASCII Characters in JSP Pages

- Characters Traveling from JSP Files to Browser Screens
- ASCII Characters in JSP Pages
- Presenting Non ASCII Characters in HTML Documents
- Entering Non ASCII Characters in Java Strings
- Java Strings - Byte Sequences Encoded for Local Languages
- Java Strings - Unicode Codes - Local Language Independent
- Entering Non ASCII Characters as Static HTML Text
- Static HTML Text - HTML Page

- Static HTML Text - JSP Page in Standard Syntax
- Static HTML Text - JSP Page in XML Syntax
- Supporting Characters from Multiple Languages

JSP Performance

- Calculating Prime Numbers
- Response Time of "Hello" Page

JSP Standard Tag Libraries (JSTL)

- What is JSTL?
- Installing JSTL 1.0 Implementation - Standard Taglib 1.0.4
- "Hello world!" with JSTL
- JSTL in XML Style JSP Pages
- JSTL Requirements

JSTL Syntax and Expression Language

- JSTL Syntax
- Expression Language
- Literal Data and Named Variables
- Basic Operators and Operations
- Implicit Objects
- Accessing Collection Elements and Object Properties
- ExpExample.jsp - Expression Example Page
- pageContext Attributes and JSTL Top Level Identifiers

JSTL - Core Library

- JJSTL Core Library
- c:out Action
- c:set Action
- c:if Action
- c:choose Action
- c:forEach Action
- c:forTokens Action
- JSTL Core Example - JstlObjects.jsp
- JSTL Core Example - JstlPrimeNumbers.jsp

JSP Custom Tag

- What is a Custom Tag?
- "Hello world!" Custom Tag
- How Custom Tag Works

JSP Tag Java Interface

- javax.servlet.jsp.tagext.* Package
- BodyTag Interface
- Implenting BodyTag Interface - TraceTag.java
- The Servlet Class - TraceTagTest_jsp.java
- Dummy Implementation of IterationTag Interface - TagSupport Class

JSP Tag Attribute Handling

- Tag Attribute Setter Method
- Tag Attribute Setter Method Example - EchoTag.java
- Tag Attribute Value Type Conversion
- Tag Attribute Value Type Conversion Example - AttValueTag.java
- Tag Attribute Value Expression
- Tag Attribute Value Expression Example - AttObjectTag.java

Tags Working Together

- Nested Tags
- Sharing Data with Other Tags

Tomcat 4.1.18 with JDK 1.4.1

- Upgrading Tomcat 4.1.18 to JDK 1.4.1
- Compilation Errors with Tomcat 4.1.18 and JDK 1.4.1
- JavaBean in a Named Package - TempratureConvertorBean.java

References

Key Words: attachment, Big5, book, Chinese, content-disposition, content_type, cookie, custom tag, debugging, example, expression language, GB2312, GBK, header lines, HTTP/1.1, HTTP Response, internationalization, i18n, IterationTag, JavaBeen, JSP, JSTL, JSTL-EL, localization, MIME, named package, online, pageContext, performance, Perl, Servlet, session, Taglib, TagSupport, tag interface, TLD, Tomcat, tutorial, unnamed package, Unicode, useBean, UTF-8, XML

About This Book

This book is a collection of notes and tutorial codes I wrote while I was leaning JSP (JavaServer Pages).

Revision history:

- Version 3.03, 2003. Minor updates.
- Version 3.00, 2003. Added notes on JSTL tags and custom tags.
- Version 2.00, 2002. Reformatted in XHTML and PDF formats.
- Version 1.00, 2001. Started in text format.

Copyright:

- The notes and example codes are Copyright © 2001 - 2003 by Dr. Herong Yang. All rights reserved.
- Using and distributing the notes and example codes are permitted, as long as the above copyright notice is retained.
- The example codes is provided as-is, with no warranty of any kind.

Tomcat

Installing Tomcat 4.1.18

Tomcat is a free, open-source implementation of Java Servlet and JavaServer Pages technologies developed under the Jakarta project at the Apache Software Foundation.

Tomcat 4.1.18 is a web server and supports Servlet 2.3 and JSP 1.2. It requires JDK 1.2 or later. I did the following to get Tomcat 4.1.18 installed:

1. Checked JDK requirement. I had JDK 1.3.1 installed on \local\jdk1.3.1.

2. Downloaded tomcat-4.1.18.zip from http://www.apache.inetcosmos.org/dist/jakarta/tomcat-4/binaries/.

3. Unziped tomcat-4.1.18.zip in \local.

4. Started Tomcat server:

```
cd \local\jakarta-tomcat-4.1.18\bin
set JAVA_HOME=\local\jdk1.3.1
startup
```

5. Tomcat created a separate command window, on which I got:

```
[INFO] Registry - -Loading registry information
[INFO] Registry - -Creating new Registry instance
[INFO] Registry - -Creating MBeanServer
[INFO] Http11Protocol - -Initializing Coyote HTTP/1.1 on port 8080
Starting service Tomcat-Standalone
Apache Tomcat/4.1.18
[INFO] Http11Protocol - -Starting Coyote HTTP/1.1 on port 8080
[INFO] ChannelSocket - -JK2: ajp13 listening on 0.0.0.0/0.0.0.0:8009
[INFO] JkMain - -Jk running ID=0 time=30/130  config=C:\local\jakarta-
tomcat-4.1.18\bin\..\conf\jk2.properties
```

6. Ran Internet Explorer (IE) with url: http://localhost:8080. I got the Tomcat home page with the following message on it:

If you're seeing this page via a web browser, it means you've setup Tomcat successfully. Congratulations!

Using Tomcat as a Web Server

To find out where it document root directory of this Web server, let's create hello.html:

```
<html><body>Hello world!</body></html>
```

Then save hello.html to \local\jakarta-tomcat-4.1.18\webapps\ROOT. Now, run IE with url: http://localhost:8080/hello.html. You should see "Hello world!" in the IE window.

Now, we know that the Web server:

```
Listening on port: 8080
Serving document from: \local\jakarta-tomcat-4.1.18\webapps\ROOT
```

To change the port number, you need to edit \local\jakarta-tomcat-4.1.18\conf\server.xml.

"Hello world!" Example

To verify if Tomcat supports JSP or not, let's create hello.jsp:

```
<html><body>
<% out.println("Hello world!"); %>
</body></html>
```

Then save hello.jsp to \local\jakarta-tomcat-4.1.18\webapps\ROOT. Now, ran IE with url: http://localhost:8080/hello.jsp. You should see "Hello world!" in the IE window.

Congratulations! I have successfully served an JSP page through Tomcat.

JavaServer Pages (JSP)

What is JSP?

JSP is a technology, not a language. It allows Web page authors to put dynamic data into a Web document with Java statements embedded in special HTML tags. The embedded Java statements will be executed by the JSP enabled Web server, not by the Web browser.

JSP Page is a Web page, HTML document, with enbedded Java statements in a structure defined by the JSP specification. Here is a JSP page example, hello.jsp:

```
<html><body>
<% out.println("Hello world!"); %>
</body></html>
```

Line 1 and 3 are normal HTML tags. But line 2 is a Java statement embedded in special HTML tag.

We know how a normal Web page is served by a normal Web server:

- Step 1: The Web browser sends a HTTP request to the Web server with the path name of the Web page.
- Step 2: The Web server picks up the Web page by following the specified path name.
- Step 3: The Web server puts the content of the Web page without any changes into a HTTP response.
- Step 4: The Web server sends the HTTP response to the Web browser.

Here is how a JSP page is served by a JSP Web server:

- Step 1: The Web browser sends a HTTP request to the Web server with the path name of the JSP page.
- Step 2: The Web server checks to see if there is a compiled version of the requested JSP page. If no, it compiles the requested JSP page immediately.
- Step 3: The Web server executes the compiled version of the requested JSP page, and collects the output of the execution.
- Step 4: The Web server puts the output of the execution into a HTTP response.
- Step 5: The Web server sends the HTTP response to the Web browser.

There are two key processes involved in serving a JSP page:

- **Compilation**: A JSP page must be compiled into a Java Servlet class, before it can be executed. The server can compile a JSP page in real-time when the page is requested for the first time, if the page is not pre-compiled.
- **Execution**: When a JSP page is requested, its compiled class will be executed on the server. The server will send back the output of the execution, not the content of the JSP page.

"Hello world!" Java Servlet Class

To understand more about the JSP compilation process, let's use my hello.jsp again:

```
<html><body>
<% out.println("Hello world!"); %>
</body></html>
```

Then save hello.jsp to \local\jakarta-tomcat-4.1.18\webapps\ROOT, and run IE with url: http://localhost:8080/hello.jsp. You should see "Hello world!" in the IE window.

Now, if you look at the directory: \local\jakarta-tomcat-4.1.18\work\standalone\localhost_, you will see a Java file: hello_jsp.java,

```
package org.apache.jsp;

import javax.servlet.*;
import javax.servlet.http.*;
import javax.servlet.jsp.*;
import org.apache.jasper.runtime.*;

public class hello_jsp extends HttpJspBase {

  private static java.util.Vector _jspx_includes;

  public java.util.List getIncludes() {
    return _jspx_includes;
  }

  public void _jspService(HttpServletRequest request,
     HttpServletResponse response)
       throws java.io.IOException, ServletException {

    JspFactory _jspxFactory = null;
    javax.servlet.jsp.PageContext pageContext = null;
    HttpSession session = null;
    ServletContext application = null;
```

```
    ServletConfig config = null;
    JspWriter out = null;
    Object page = this;
    JspWriter _jspx_out = null;

    try {
      _jspxFactory = JspFactory.getDefaultFactory();
      response.setContentType("text/html;charset=ISO-8859-1");
      pageContext = _jspxFactory.getPageContext(this, request, response,
                  null, true, 8192, true);
      application = pageContext.getServletContext();
      config = pageContext.getServletConfig();
      session = pageContext.getSession();
      out = pageContext.getOut();
      _jspx_out = out;

      out.write("<html>");
      out.write("<body>\r\n");
  out.println("Hello world!");
      out.write("\r\n");
      out.write("</body>");
      out.write("</html>");
    } catch (Throwable t) {
      out = _jspx_out;
      if (out != null && out.getBufferSize() != 0)
        out.clearBuffer();
      if (pageContext != null) pageContext.handlePageException(t);
    } finally {
      if (_jspxFactory != null)
        _jspxFactory.releasePageContext(pageContext);
    }
  }
}
```

You will also see a Java class file: hello_jsp.class.

What happened here was that Tomcat, the JSP Web server, has translated hello.jsp into hello_jsp.java, and compiled it to hello_jsp.class.

The Java file, hello_jsp.java, shows that:

- hello_jsp is a sub class of org.apache.jasper.runtime.HttpJspBase, which is a sub class of javax.servlet.http.HttpServlet.
- The important method in hello_jsp is _jspService() with two objects listed as parameters: one represents the Servlet request, and the other represents the Servlet response.

- The static content of hello.jsp is translated into out.write() statements.
- The embedded Java statement in hello.jsp is copied directly.

Writing JSP Servlet Class Directly

Now we know that a JSP page is served by the JSP Web server by executing the JSP Servlet Java class translated from the JSP page. This means that we can write a JSP Servlet Java class directly, and ask the JSP Web server to serve it.

To try this idea, let's first write this JSP page, fake.jsp, and save it to \local\jakarta-tomcat-4.1.18\webapps\ROOT:

```
<!--
 - fake.jsp
 - Copyright (c) 2002 by Dr. Herong Yang. All rights reserved.
-->
<html><body>
This a faked JSP page. The real content will come from the output
of the JSP Servlet class.
</body></html>
```

Then, write the following JSP Servlet class, fake_jsp.java: and save it to \local\jakarta-tomcat-4.1.18\work\standalone\localhost_:

```
/**
 * fake_jsp.java
 * Copyright (c) 2002 by Dr. Herong Yang. All rights reserved.
 */
package org.apache.jsp;
import javax.servlet.*;
import javax.servlet.http.*;
import javax.servlet.jsp.*;
import org.apache.jasper.runtime.*;
public class fake_jsp extends HttpJspBase {
  public java.util.List getIncludes() {
    return null;
  }
  public void _jspService(HttpServletRequest request,
    HttpServletResponse response)
    throws java.io.IOException, ServletException {
    JspFactory _jspxFactory = null;
    javax.servlet.jsp.PageContext pageContext = null;
    JspWriter out = null;
    try {
```

```
      _jspxFactory = JspFactory.getDefaultFactory();
      response.setContentType("text/html;charset=ISO-8859-1");
      pageContext = _jspxFactory.getPageContext(this, request, response,
        null, true, 8192, true);
      out = pageContext.getOut();
      out.write("<html>");
      out.write("<body>\r\n");
      out.println("Hello world! - From Servlet");
      out.write("\r\n");
      out.write("</body>");
      out.write("</html>");
    } catch (Throwable t) {
      if (out != null && out.getBufferSize() != 0)
        out.clearBuffer();
      if (pageContext != null) pageContext.handlePageException(t);
    } finally {
      if (_jspxFactory != null)
        _jspxFactory.releasePageContext(pageContext);
    }
  }
}
```

Compile this class with JDK 1.3.1:

```
cd \local\jakarta-tomcat-4.1.18\work\standalone\localhost\_
set classpath=..\..\..\..\common\lib\servlet.jar
set classpath=%classpath%;..\..\..\..\common\lib\servlet.jar
\local\jdk1.3.1\bin\javac fake_jsp.java
```

Now, run IE with url: http://localhost:8080/fake.jsp. Guess what you will get on the IE window? The text from the fake.jsp page, or the output of fake_jsp.java?

You should see the output of fake_jsp.java. Tomcat has been fooled by the file names and time stamps. When Tomcat receives a HTTP request for fake.jsp, it will look for fake_jsp.class at the JSP Servlet directory. Since fake_jsp.class is there and has newer time stamp than fake.jsp, it will assume fake_jsp.class is the latest class translated from fake.jsp, and execute it immediately.

Be aware that if you modify fake.jsp and save it back. The next time when Tomcat receives a request for fake.jsp, it will translate the new fake.jsp and replace both fake_jsp.java and fake_jsp.class. The original fake_jsp.java will be gone.

Execution Context

Predefined Objects

Now we know that the Java statements embedded in JSP pages will translated into _jspService() method of a special Servlet class. In that method, there are a couple of pre-defined objects ready for the embedded Java statements:

- out: The output stream to collect dynamic data to be mixed into the final Web document.
- this: The instance of the special Servlet class.
- request: An HttpServletRequest object representing the request received from the Web browser.
- response: An HttpServletResponse object representing the response to be delivered back to the Web browser.
- session: An HttpSession object representing the concept of linking multiple trips of requests and response into a single process unit.
- application: A ServletContext object representing the concept of grouping Servlets into a single application.
- config: A ServletConfig object.
- pageContext: A PageContext object.

The following JSP page will show you more details about those pre-defined objects:

```
<!--
 - ContextInfo.jsp
 - Copyright (c) 2002 by Dr. Herong Yang. All rights reserved.
-->
<html><body>
<p>
<b>JSP Page Context Information</b><br/>
</p>
<p>
<b>Pre-defined objects:</b><br/>
<%
  out.println("out: "+out.getClass().getName()+"<br/>");
  out.println("this: "+this.getClass().getName()+"<br/>");
  out.println("request: "+request.getClass().getName()+"<br/>");
  out.println("response: "+response.getClass().getName()+"<br/>");
```

```
  out.println("session: "+session.getClass().getName()+"<br/>");
  out.println("application: "+application.getClass().getName()
    +"<br/>");
  out.println("config: "+config.getClass().getName()+"<br/>");
  out.println("pageContext: "+pageContext.getClass().getName()
    +"<br/>");
%>
</p>
<p>
<b>Information about session:</b><br/>
<i>= pageContext.getSession();</i><br/>
<%
  out.println("Class Name: "+session.getClass().getName()+"<br/>");
  out.println("Session ID: "+session.getId()+"<br/>");
  java.util.Date d = new java.util.Date();
  d.setTime(session.getCreationTime());
  out.println("Create Time: "+d.toString()+"<br/>");
  d.setTime(session.getLastAccessedTime());
  out.println("Last Access Time: "+d.toString()+"<br/>");
  out.println("Is Session New: "+session.isNew()+"<br/>");
%>
</p>
<p>
<b>Information about sessionContext:</b><br/>
<i>= session.getSessionContext();</i><br/>
<%
  javax.servlet.http.HttpSessionContext c
    = session.getSessionContext();
  out.println("Class name: "+c.getClass().getName()+"<br/>");
%>
</p>
<p>
<b>Information about application:</b><br/>
<i>= pageContext.getServletContext();</i><br/>
<%
  out.println("Class Name: "+application.getClass().getName()+"<br/>");
  out.println("Major Version: "+application.getMajorVersion()+"<br/>");
  out.println("Minor Version: "+application.getMinorVersion()+"<br/>");
  out.println("Server Info: "+application.getServerInfo()+"<br/>");
  out.println("Serlet Context Name: "
    +application.getServletContextName()+"<br/>");
  java.util.Enumeration e = application.getServletNames();
  while (e.hasMoreElements()) {
    String n = (String) e.nextElement();
```

```
      out.println("Servlet Name: "+n+"<br/>");
    }
    e = application.getInitParameterNames();
    while (e.hasMoreElements()) {
      String n = (String) e.nextElement();
      out.println("Init Parameter Name: "+n+"<br/>");
    }
  %>
  </p>
  </body></html>
```

Output:

```
 JSP Page Context Information

 Pre-defined objects:
 out: org.apache.jasper.runtime.JspWriterImpl
 this: org.apache.jsp.ContextInfo_jsp
 request: org.apache.coyote.tomcat4.CoyoteRequestFacade
 response: org.apache.coyote.tomcat4.CoyoteResponseFacade
 session: org.apache.catalina.session.StandardSessionFacade
 application: org.apache.catalina.core.ApplicationContextFacade
 config: org.apache.catalina.core.StandardWrapperFacade
 pageContext: org.apache.jasper.runtime.PageContextImpl

 Information about session:
 = pageContext.getSession();
 Class Name: org.apache.catalina.session.StandardSessionFacade
 Session ID: 35466D59BF54A551BFBABA22B61A66EB
 Create Time: Sun Dec 22 13:40:55 EST 2002
 Last Access Time: Sun Dec 22 13:40:55 EST 2002
 Is Session New: true

 Information about sessionContext:
 = session.getSessionContext();
 Class name: org.apache.catalina.session.StandardSessionContext

 Information about application:
 = pageContext.getServletContext();
 Class Name: org.apache.catalina.core.ApplicationContextFacade
 Major Version: 2
 Minor Version: 3
 Server Info: Apache Tomcat/4.1.18
 Serlet Context Name: Welcome to Tomcat
```

A new session will be established, if this JSP page is requested for the first time. Subsequent requests will share the same session. Click the refresh button on the Web browser, you will see that the session ID will be the same, the create time will be the same, but the last access time will be the new time, and "is session new" will be "false".

The "request" Object

The "request" object is an important object to the JSP page, because it contains a lot of useful information. The following JSP page will you some details of the "request" object:

```
<!--
 - RequestInfo.jsp
 - Copyright (c) 2002 by Dr. Herong Yang. All rights reserved.
-->
<html><body>
<p>
<b>Information about request:</b><br/>
<%
  out.println("Class Name: "+request.getClass().getName()+"<br/>");
  out.println("Auth Type: "+request.getAuthType()+"<br/>");
  out.println("Context Path: "+request.getContextPath()+"<br/>");
  out.println("Method: "+request.getMethod()+"<br/>");
  out.println("Path Info: "+request.getPathInfo()+"<br/>");
  out.println("Path Translated: "+request.getPathTranslated()+"<br/>");
  out.println("Query String: "+request.getQueryString()+"<br/>");
  out.println("Remote User: "+request.getRemoteUser()+"<br/>");
  out.println("Requested Session ID: "+request.getRequestedSessionId()
    +"<br/>");
  out.println("Request URI: "+request.getRequestURI()+"<br/>");
  out.println("Request URL: "+request.getRequestURL()+"<br/>");
  out.println("Servlet Path: "+request.getServletPath()+"<br/>");
  out.println("Cookies:<br/>");
  javax.servlet.http.Cookie[] cookies = request.getCookies();
  for (int i=0; i<cookies.length; i++) {
    out.println("   "+cookies[i].getName()+": "
      +cookies[i].getValue()+"<br/>");
  }
  out.println("Headers:<br/>");
  java.util.Enumeration e = request.getHeaderNames();
  while (e.hasMoreElements()) {
    String n = (String) e.nextElement();
    out.println("   "+n+": "+request.getHeader(n)+"<br/>");
  }
```

```
%>
</p>
</body></html>
```

Output:

```
Information about request:
Class Name: org.apache.coyote.tomcat4.CoyoteRequestFacade
Auth Type: null
Context Path:
Method: GET
Path Info: null
Path Translated: null
Query String: null
Remote User: null
Requested Session ID: 13190484CD4CE195C9434A318D46950E
Request URI: /RequestInfo.jsp
Request URL: http://localhost:8080/RequestInfo.jsp
Servlet Path: /RequestInfo.jsp
Cookies:
  JSESSIONID: 13190484CD4CE195C9434A318D46950E
Headers:
  accept: image/gif, image/x-xbitmap, image/jpeg, image/pjpeg, ...
  accept-language: en-us
  accept-encoding: gzip, deflate
  user-agent: Mozilla/4.0 (compatible; MSIE 6.0; MSNIA; Windows NT ...
  host: localhost:8080
  connection: Keep-Alive
  cookie: JSESSIONID=13190484CD4CE195C9434A318D46950E
```

The "session" Object

session: A object provided by the JSP server to hold information and methods common to all JSP pages running under one session. The session object must be an instance of a class that implements the javax.servlet.http.HttpSession interface defined by the J2EE specification. Here is the highlights of the HttpSession interface defined in J2EE 1.3:

- **getAttribute()**: Returns the object that is associated to the specified key string. defined in the session.
- **getAttributeNames()**: Returns an Enumeration object that contains all the key strings defined in the session.
- **getCreationTime()**: Returns the time when this session was created, measured in milliseconds since midnight January 1, 1970 GMT.
- **getId()**: Returns the session ID as a string.

- **setAttribute()**: Associate an object with the specified key string and store them to this session.

More information about the session object and sample JSP codes will be give in another chapter.

JSP Elements

Syntactic Elements of a JSP Page

There are two types of data in a JSP page:

- Template Data: The static part, anything that will be copied directly to the response by the JSP server.
- JSP Elements: The dynamic part, anything that will be translated and executed by the JSP server.

There are three types of JSP elements:

Directive Element: A JSP element that provides global information for the translation phase. There are two ways to write a directive element:

<%@ directive_name attribute=value ... %>

Action Element: A JSP element that provides information for the execution phase.

<action_name attribute=value ...>action_body</action_name>
<action_name attribute=value .../>

Scripting Element: A JSP element that provides embedded Java statements. There are three types of scripting elements:

Declaration Element: A JSP element that provides the embedded Java declaration statements to be inserted into the Servlet class.

<%! Java decalaration statements %>

Scriptlet Element: A JSP element that provides the embedded Java statements to be executed as part of the service method of the Servlet class. There are two ways to write a scriptlet element:

<% Java statements %>

Expression Element: A JSP element that provides the embedded Java expressions to be

evaluated as part of the service method of the Servlet class. There are two ways to write an express element:

<% Java expressoins %>

Writing JSP Pages in XML Format

JSP pages can also be written in XML format. To do this, you have to use the XML version of the syntaxes for directive elements, declaration, scriptlet and expression elements:

<jsp:directive.directive_name attribute=value .../>
<jsp:declaration> Java decalaration statements </jsp:declaration>
<jsp:scriptlet> Java statements </jsp:scriptlet>
<jsp:expression> Java expressions </jsp:expression>

You also have to create a "jsp:root" element. Here is my "Hello world!" example in XML format:

```
<?xml version="1.0"?>
<jsp:root xmlns:jsp="http://java.sun.com/JSP/Page" version="1.2">
<!-- hello_xml.jsp
    Copyright (c) 2002 by Dr. Herong Yang
-->
<html><body>
<jsp:scriptlet>out.println("Hello world!");</jsp:scriptlet>
</body></html>
</jsp:root>
```

Then open this JSP page with IE, what you will see is this:

```
- <html>
  <body>Hello world!</body>
  </html>
```

What happens here is that the server is probably sending response with content-type set to text/xml. To fix the problem, we need to use a directive element to set the contect-type back to text/html. Here is the modified JSP page, hello_xml_html.jsp:

```
<?xml version="1.0"?>
<jsp:root xmlns:jsp="http://java.sun.com/JSP/Page" version="1.2">
<!-- hello_xml_html.jsp
    Copyright (c) 2002 by Dr. Herong Yang
-->
```

```
<jsp:directive.page contentType="text/html"/>
<html><body>
<jsp:scriptlet>out.println("Hello world!");</jsp:scriptlet>
</body></html>
</jsp:root>
```

There is another problem with the XML format of JSP pages. The problem is caused by the XML requirement that the entire file must be well formed. This is a tough requirement for the embedded Java statements, because of the '<' operator and other special operators. So we need to use the <![CDATA[...]]> to protect the Java statements. Here is an example, LoopTest.jsp:

```
<?xml version="1.0"?>
<jsp:root xmlns:jsp="http://java.sun.com/JSP/Page" version="1.2">
<!--
 - LoopTest.jsp
 - Copyright (c) 2002 by Dr. Herong Yang. All rights reserved.
-->
<jsp:directive.page contentType="text/html"/>
<html><body>
<p>
<jsp:scriptlet><![CDATA[
  out.println("<b>Sum of 1, 2, ... 10:</b><br/>");
  int sum = 0;
  for (int i=0; i<10; i++) {
    sum += i+1;
  }
  out.println(sum);
]]></jsp:scriptlet>
</p>
</body></html>
</jsp:root>
```

If you open this page with IE, you should have no problem to get the result.

Directive Elements

The most commonly used directive is probably the "page" directive, which specifies page level properties for the JSP server. Here is some examples:

```
<jsp:directive.page import="java.util.*"/>
<jsp:directive.page import="java.net.*" session="false"/>
<jsp:directive.page contentType="text/html"/>
```

Note that:

- All directive attributes are optional.
- "import" tells the JSP server to insert "import" statements into the Servlet class.
- "session" tells the JSP server to execute the Servlet with a session or without a session. The default value is "true".
- "contentType" tells the JSP server to set content-type of HTTP response with the specified value.

Another commonly used directive is the "include" directive, which tells the ASP server to include the specified JSP page into the current JSP page before translation.

```
<jsp:directive.include file="header.jsp"/>
<jsp:directive.include file="footer.jsp"/>
```

Action Elements

The most commonly used action is probably the "include" action, which tells the ASP server to execute the specified JSP page and include the output into the current JSP page.

```
<jsp:include page="CurrentMonthCalendar.jsp"/>
```

Another commonly used action is the "useBean" action, which loads a JavaBean object into the JSP page.

```
<jsp:useBean id="object_name" class="class_name"/>
```

JSP Example - "CurrentTime.jsp"

The following example has three JSP files working together to show you how to use "decalaration" elements, "include" directive elements and "include" action elements. Here is the main JSP file, CurrentTime.jsp:

```
<?xml version="1.0"?>
<jsp:root xmlns:jsp="http://java.sun.com/JSP/Page" version="1.2">
<!-- CurrentTime.jsp
   Copyright (c) 2002 by Dr. Herong Yang
-->
<jsp:directive.page contentType="text/html"/>
<html><body>
<jsp:directive.page import="java.util.*"/>
<jsp:directive.page import="java.text.*"/>
```

```
<jsp:declaration>
  private Date now;
  private JspWriter out;
  private void printTime(String tz) throws Throwable {
    DateFormat df = DateFormat.getInstance();
    df.setTimeZone(TimeZone.getTimeZone(tz));
    out.println(tz+": "+df.format(now)+"&lt;br/>");
  }
</jsp:declaration>
<p>
<b>Current time in different time zones:</b><br/>
<jsp:scriptlet>
  this.out = out;
  now = new Date();
  printTime("America/New_York");
  printTime("America/Los_Angeles");
  printTime("Asia/Shanghai");
  printTime("Europe/Paris");
  printTime("Europe/Moscow");
</jsp:scriptlet>
</p>
<p>
<jsp:directive.include file="JvmStamp.jsp"/>
</p>
<p>
<jsp:include page="TimeStamp.jsp"/>
</p>
</body></html>
</jsp:root>
```

Here is the JSP file used by the include directive element, JvmStamp.jsp:

```
<?xml version="1.0"?>
<jsp:root xmlns:jsp="http://java.sun.com/JSP/Page" version="1.2">
<!-- JvmStamp.jsp
    Copyright (c) 2002 by Dr. Herong Yang
-->
<b>Current JVM:</b><br/>
<jsp:scriptlet>
  String s;
  s = "java.vm.name";
  out.println(s+": "+System.getProperty(s)+"&lt;br/>");
  s = "java.vm.version";
  out.println(s+": "+System.getProperty(s)+"&lt;br/>");
```

```
   s = "os.name";
   out.println(s+": "+System.getProperty(s)+"&lt;br/>");
   s = "os.version";
   out.println(s+": "+System.getProperty(s)+"&lt;br/>");
 </jsp:scriptlet>
 </jsp:root>
```

Here is the JSP file used by the include action element, TimeStamp.jsp:

```
 <?xml version="1.0"?>
 <jsp:root xmlns:jsp="http://java.sun.com/JSP/Page" version="1.2">
 <!-- TimeStamp.jsp
     Copyright (c) 2002 by Dr. Herong Yang
 -->
 <b>Current Time: </b>
 <jsp:scriptlet>
   java.util.Date d = new java.util.Date();
   out.println(d.toString());
 </jsp:scriptlet>
 </jsp:root>
```

Here is what I got by requesting CurrentTime.jsp from IE:

```
 Current time in different time zones:
 America/New_York: 12/23/02 9:38 PM
 America/Los_Angeles: 12/23/02 6:38 PM
 Asia/Shanghai: 12/24/02 10:38 AM
 Europe/Paris: 12/24/02 3:38 AM
 Europe/Moscow: 12/24/02 5:38 AM

 Current JVM:
 java.vm.name: Java HotSpot(TM) Client VM
 java.vm.version: 1.3.1_01
 os.name: Windows 2000
 os.version: 5.0

 Current Time: Mon Dec 23 21:38:10 EST 2002
```

It's very interesting to see the Servlet class translated from CurrentTime.jsp:

```
 package org.apache.jsp;
 import javax.servlet.*;
 import javax.servlet.http.*;
 import javax.servlet.jsp.*;
```

```
import org.apache.jasper.runtime.*;
import java.util.*;
import java.text.*;

public class CurrentTime_jsp extends HttpJspBase {

  private Date now;
  private JspWriter out;
  private void printTime(String tz) throws Throwable {
     DateFormat df = DateFormat.getInstance();
     df.setTimeZone(TimeZone.getTimeZone(tz));
     out.println(tz+": "+df.format(now)+"<br/>");
  }

  private static java.util.Vector _jspx_includes;

  static {
   _jspx_includes = new java.util.Vector(1);
   _jspx_includes.add("/jsp/JvmStamp.jsp");
  }

  public java.util.List getIncludes() {
   return _jspx_includes;
  }

  public void _jspService(HttpServletRequest request,
       HttpServletResponse response)
       throws java.io.IOException, ServletException {

   JspFactory _jspxFactory = null;
   javax.servlet.jsp.PageContext pageContext = null;
   HttpSession session = null;
   ServletContext application = null;
   ServletConfig config = null;
   JspWriter out = null;
   Object page = this;
   JspWriter _jspx_out = null;

   try {
     _jspxFactory = JspFactory.getDefaultFactory();
     response.setContentType("text/html;charset=UTF-8");
     pageContext = _jspxFactory.getPageContext(this, request, response,
                 null, true, 8192, true);
     application = pageContext.getServletContext();
```

```
   config = pageContext.getServletConfig();
   session = pageContext.getSession();
   out = pageContext.getOut();
   _jspx_out = out;

   out.write("<html>");
   out.write("<body>");
   out.write("<p>");
   out.write("<b>");
   out.write("Current time in different time zones:");
   out.write("</b>");
   out.write("<br/>");

 this.out = out;
 now = new Date();
 printTime("America/New_York");
 printTime("America/Los_Angeles");
 printTime("Asia/Shanghai");
 printTime("Europe/Paris");
 printTime("Europe/Moscow");
   out.write("</p>");
   out.write("<p>");
   out.write("<b>");
   out.write("Current JVM:");
   out.write("</b>");
   out.write("<br/>");

 String s;
 s = "java.vm.name";
 out.println(s+": "+System.getProperty(s)+"<br/>");
 s = "java.vm.version";
 out.println(s+": "+System.getProperty(s)+"<br/>");
 s = "os.name";
 out.println(s+": "+System.getProperty(s)+"<br/>");
 s = "os.version";
 out.println(s+": "+System.getProperty(s)+"<br/>");
   out.write("</p>");
   out.write("<p>");
   JspRuntimeLibrary.include(request, response, "TimeStamp.jsp", out,
     false);
   out.write("</p>");
   out.write("</body>");
   out.write("</html>");
  } catch (Throwable t) {
```

```
      out = _jspx_out;
      if (out != null && out.getBufferSize() != 0)
        out.clearBuffer();
      if (pageContext != null) pageContext.handlePageException(t);
    } finally {
      if (_jspxFactory != null)
        _jspxFactory.releasePageContext(pageContext);
    }
  }
}
```

Couple of interesting things to know in this example:

- The pre-defined objects like "out" is not available in the directive elements. So I used an instance variable "out" to pass the pre-defined "out" to my "printTime()" method in the declaration element.
- Since I am using XML format, "<" needs to be replaced by the entity name "<" in any Java statements. During the translation process, the entity names will be replaced back to "<".
- No Servlet class was created for JvmStamp.jsp, because its content was included into CurrentTime.jsp during the translation process.
- "TimeStamp.jsp" was treated differently than "JvmStamp.jsp". "TimeStamp.jsp" was translated independently into a Servlet class. Its service method was called by CurrentTime_jsp.java during execution.

JSP Sessions and Debugging

This chapter describes:

- What is a session, and how a JSP server uses cookies to pass an IDs to the browser to link multiple HTTP requests together as a session.
- What is the session object, and what functions are available on the session object.
- A sample application to show you how to pass data between JSP pages.
- How Perl tools can be used to help debug JSP applications at the HTTP communication level.

What Is a Session?

session: A concept to represent a series of HTTP requests and responses exchanged between a specific Web browser and a specific Web server, see the following diagram:

```
          Server              Browser
ID created  | <-- Request #1  --- |
            | --- Response #1 --> | ID kept as cookie
            | <-- Request #2  --- | ID send back to server
            | --- Response #2 --> |
            | <-- Request #3  --- | ID send back to server
            | --- Response #3 --> |
            |    ......          |
```

The session concept is managed by the server. When the first request comes from a browser on a new host, the server makes the beginning of a new session, and assigns a new session ID. The session ID will be then send to the browser as cookie. The browser will remember this ID, and send the ID back to the server in the subsequent requests. When the server receives a request with session ID in them, it knows this is a continuation of an existing session.

When the server receives a request from a browser on a new host (request without a session ID), the server not only creates a new session ID, it also creates a new session object associated with the session ID. See the next section for details.

If there is no subsequent request coming back for a long time for a particular session ID, that session will be timed out. After the session has been timed out, if the browser comes back again with the associated session ID, the server will give an invalid session error.

The "session" Object

session: A object provided by the JSP server to hold information and methods common to all JSP pages running under one session. The session object must be an instance of a class that implements the javax.servlet.http.HttpSession interface defined by the J2EE specification. Here is the highlights of the HttpSession interface defined in J2EE 1.3:

- **getAttribute()**: Returns the object that is associated to the specified key string. defined in the session.
- **getAttributeNames()**: Returns an Enumeration object that contains all the key strings defined in the session.
- **getCreationTime()**: Returns the time when this session was created, measured in milliseconds since midnight January 1, 1970 GMT.
- **getId()**: Returns the session ID as a string.
- **getLastAccessedTime()**: Returns the last time the client sent a request associated with this session, as the number of milliseconds since midnight January 1, 1970 GMT.
- **getMaxInactiveInterval()**: Returns the maximum time interval, in seconds, that the servlet container will keep this session open between client accesses.
- **removeAttribute()**: Removes the object associated with the specified key string from this session.
- **setAttribute()**: Associate an object with the specified key string and store them to this session.
- **setMaxInactiveInterval()**: Specifies the time, in seconds, between client requests before the servlet container will invalidate this session.

Passing Values between Pages

There are many ways to pass values from one pages to the next pages:

- Putting values into the session object.
- Putting values into the application object
- Putting values at the end of the redirect URL.

In the following example, I have two JSP pages working together as a registration process. Here is the fist JSP page, RegForm.jsp:

```
<?xml version="1.0"?>
<jsp:root xmlns:jsp="http://java.sun.com/JSP/Page" version="1.2">
<!-- RegForm.jsp
    Copyright (c) 2002 by Dr. Herong Yang
-->
<jsp:directive.page contentType="text/html"/>
<jsp:declaration><![CDATA[
```

```
  private String getItem(String queryString, String key) {
    String value = null;
    if (queryString!=null) {
      int i = queryString.indexOf(key);
      if (i>-1) {
        i = i + key.length();
        int j = queryString.indexOf("&",i);
        if (j>-1) {
          value = queryString.substring(i,j);
        } else {
          value = queryString.substring(i);
        }
        if (value.startsWith("=")) {
          value = value.replaceFirst("=","");
        }
      }
    }
    return value;
  }
]]></jsp:declaration>
<jsp:scriptlet><![CDATA[
  String lastUser = (String) application.getAttribute("name");
  if (lastUser==null) {
    lastUser = "Nobody";
    application.setAttribute("name",lastUser);
  }
  String queryString = request.getQueryString();
  String submit = getItem(queryString,"submit");
  if (submit!=null && submit.equals("Submit")) {
  // Collecting the input data
    session.setAttribute("name",getItem(queryString,"name"));
    session.setAttribute("pass",getItem(queryString,"pass"));
    application.setAttribute("name",getItem(queryString,"name"));
    response.sendRedirect("RegDone.jsp?color="
      +getItem(queryString,"color"));
  } else {
  // Presenting the registration form
    out.print("<html><body>");
    out.print("<b>Registration Form</b>:<br/>");
    out.print("<form action=RegForm.jsp method=get>");
    out.print("Login Name:");
    out.print("<input type=text size=16 name=name><br/>");
    out.print("Password:");
    out.print("<input type=text size=16 name=pass><br/>");
```

```
      out.print("Favor Color:");
      out.print("<input type=text size=16 name=color><br/>");
      out.print("<input type=submit name=submit value=Submit></br>");
      out.print("</form>");
      out.print("Your session ID is "+session.getId()+"<br/>");
      out.print("Last user on the server: "+lastUser+"<br/>");
      out.print("</body></html>");
    }
 ]]></jsp:scriptlet>
 </jsp:root>
```

Here is the second JSP page, RegDone.jsp:

```
 <?xml version="1.0"?>
 <jsp:root xmlns:jsp="http://java.sun.com/JSP/Page" version="1.2">
 <!-- RegDone.jsp
     Copyright (c) 2002 by Dr. Herong Yang
 -->
 <jsp:directive.page contentType="text/html"/>
 <jsp:declaration><![CDATA[
   private String getItem(String queryString, String key) {
     String value = null;
     if (queryString!=null) {
       int i = queryString.indexOf(key);
       if (i>-1) {
         i = i + key.length();
         int j = queryString.indexOf("&",i);
         if (j>-1) {
           value = queryString.substring(i,j);
         } else {
           value = queryString.substring(i);
         }
         if (value.startsWith("=")) {
           value = value.replaceFirst("=","");
         }
       }
     }
     return value;
   }
 ]]></jsp:declaration>
 <jsp:scriptlet><![CDATA[
   out.print("<html><body>");
   String lastUser = (String) application.getAttribute("name");
   String queryString = request.getQueryString();
```

```
   out.print("<b>Thank you for registering with us</b>:<br/>");
   out.print("Login Name: "+(String)session.getAttribute("name")
     +"<br/>");
   out.print("Password: "+(String)session.getAttribute("pass")
     +"<br/>");
   out.print("Favor Color: "+getItem(queryString,"color")+"<br/>");
   out.print("Your session ID is "+session.getId()+"<br/>");
   out.print("Last user on the server: "+lastUser+"<br/>");
   out.print("</body></html>");
 ]]></jsp:scriptlet>
 </jsp:root>
```

Request RegForm.jsp with a browser, you will get a page similar to this:

Registration Form:

Login Name:
Password:
Favor Color:

Your session ID is 2B20E475CA7B0FFC4C2E752ABF24C772
Last user on the server: Nobody

Then fill in the page with

Login Name: Herong
Password: Secret
Favor Color: Red

Click the Submit button, you will get the output of RegDone.jsp page:

Thank you registrating with us:
Login Name: Herong
Password: Secret
Favor Color: Red
Your session ID is 2B20E475CA7B0FFC4C2E752ABF24C772
Last user on the server: Herong

Then close your browser, and start it again with RegForm.jsp, you will get:

Registration Form:

Login Name:

Password:
Favor Color:

Your session ID is A497631211582DE3799223EEF31BCF4F
Last user on the server: Herong

A number of interesting notes:

- RegForm.jsp page is designed to serve two functions: presenting the form and collecting data from the submitted form.
- When RegForm.jsp is requested for the first time, there will be no "submit" in the query string. So the JSP code will continue with the presenting-form section.
- When the user finishes filling in the form, and clicks the Submit button, the browser will request RegForm.jsp again and attach all the data in the form as the query string. This behavior is specified by the <form> tag.
- When RegForm.jsp is requested by the Submit button, "submit" will have "Submit" as its value. So the JSP code will continue with the collecting-data section.
- In the collecting-data section, I want to pass the collected data to another JSP page, RegDone.jsp. Here I use two approaches to pass data to RegDone.jsp.
- "name" and "pass" are passed through the session object. This is probably the best approach to pass data from one JSP page to another.
- "color" is passed as part of the redirect URL. It will show up in the browser's URL area. So you should not use this approach to pass sensitive information from one JSP page to another.
- The application object is also used to pass "name" from RegForm.jsp to RegDone.jsp. This copy of "name" is used as the "last user on the server", which is not session specific. In general, passing session specific data through the application object is not safe. Different sessions could override each other.
- When the browser is closed and started again, a new session object is created but the application is still the same. This is why you see "Herong" again as the last user on the server.

HTTP Communication Level Debugging

If you have a problem with your JSP application at the HTTP communication level, one good debugging tool is the Perl LWP package. It can be used as a Web browser to talk to your JSP application, and to log everything at the HTTP communication level.

Here is my sample Perl program, reg_client.pl, designed to work with my previous JSP registration application:

```
#- reg_client.pl
#- Copyright (c) 2002 by Dr. Herong Yang
  use LWP::Debug qw(+);
```

```
  use LWP::UserAgent;
  use HTTP::Cookies;
  ($url) = @ARGV;
  $url =  'http://localhost' unless $url;
  $ua = new LWP::UserAgent;
  $cookie_jar = HTTP::Cookies->new;
  &getForm();
  &submitForm();
  exit;
 sub getForm {
  $u =  $url.'/RegForm.jsp';
  my $req = new HTTP::Request GET => $u;
  my $res = $ua->request($req);
  $req = $res->request();
  $cookie_jar->extract_cookies($res);
  &dump($req,$res);
 }
 sub submitForm {
  $u =  $url.'/RegForm.jsp?name=Mike&pass=None&color=Blue&submit=Submit';
  my $req = new HTTP::Request GET => $u;
  $cookie_jar->add_cookie_header($req);
  my $res = $ua->request($req);
  $req = $res->request();
  $cookie_jar->extract_cookies($res);
  &dump($req,$res);
 }
 sub dump {
    local ($req,$res) = @_;
    print "\nREQUEST-HEADERS\n";
    print $req->headers_as_string();
    print "\nREQUEST-CONTENT\n";
    print $req->content;
    if ($res->is_success) {
       print "\nRESPONSE-HEADERS\n";
       print $res->headers_as_string();
       print "\nRESPONSE-CONTENT\n";
       print $res->content;
    } else {
       print "\nRESPONSE-ERROR\n";
       print $res->error_as_HTML();
    }
 }
```

A couple of notes to help you to understand this program:

- "use LWP::Debug qw(+);" turns on the debugging at the highest level.
- A "LWP::UserAgent" object is used to send a HTTP request to the HTTP server.
- "HTTP:Request" objects are used to compose HTTP requests.
- "$cookie_jar->extract_cookies($res);" is used to extract cookies from the response. This is very important, because JSP server is sending the session ID as a cookie to the client and expecting the client to send it back in the next request.
- "$cookie_jar->add_cookie_header($req);" is used to add the cookies received from the previous response to the current request. One of the cookies is the session id, which is important for the JSP server to recognize the current request is a continuation of the previous request.

If you run it with "reg_client.pl http://localhost:8080 > client.out" in a command window, you will get the following in the window:

```
LWP::UserAgent::new: ()
LWP::UserAgent::request: ()
LWP::UserAgent::simple_request: GET http://localhost:8080/RegForm.jsp
LWP::UserAgent::_need_proxy: Not proxied
LWP::Protocol::http::request: ()
LWP::Protocol::http::request: GET /RegForm.jsp HTTP/1.0
Host: localhost:8080
User-Agent: libwww-perl/5.51

LWP::Protocol::http::request: reading response
LWP::Protocol::http::request: HTTP/1.1 200 OK
Set-Cookie: JSESSIONID=887896C93DFFF372EB38818BF9F68DB2; Path=/
Content-Type: text/html;charset=UTF-8
Content-Length: 393
Date: Sat, 28 Dec 2002 20:37:04 GMT
Server: Apache Coyote/1.0
Connection: close

<html><body><b>Registration Form</b>:<br/><form action=RegForm.jsp met
hod=get>Login Name:<input type=text size=16 name=name><br/>Password:<i
nput type=text size=16 name=pass><br/>Favor Color:<input type=text siz
e=16 name=color><br/><input type=submit name=submit value=Submit></br>
</form>Your session ID is 887896C93DFFF372EB38818BF9F68DB2<br/>Last us
er on the server: Nobody<br/></body></html>
LWP::Protocol::http::request: HTTP/1.1 200 OK
LWP::Protocol::collect: read 393 bytes
LWP::UserAgent::request: Simple response: OK
HTTP::Cookies::extract_cookies: Set cookie JSESSIONID => 887896C93DFFF
372EB38818BF9F68DB2
HTTP::Cookies::add_cookie_header: Checking localhost.local for cookies
```

```
HTTP::Cookies::add_cookie_header: - checking cookie path=/
HTTP::Cookies::add_cookie_header:  - checking cookie JSESSIONID=887896
C93DFFF372EB38818BF9F68DB2
HTTP::Cookies::add_cookie_header:    it's a match
HTTP::Cookies::add_cookie_header: Checking .local for cookies
LWP::UserAgent::request: ()
LWP::UserAgent::simple_request: GET http://localhost:8080/RegForm.jsp?
name=Mike&pass=None&color=Blue&submit=Submit
LWP::UserAgent::_need_proxy: Not proxied
LWP::Protocol::http::request: ()
LWP::Protocol::http::request: GET /RegForm.jsp?name=Mike&pass=None&col
or=Blue&submit=Submit HTTP/1.0
Host: localhost:8080
User-Agent: libwww-perl/5.51
Cookie: JSESSIONID=887896C93DFFF372EB38818BF9F68DB2
Cookie2: $Version=1

LWP::Protocol::http::request: reading response
LWP::Protocol::http::request: HTTP/1.1 302 Moved Temporarily
Location: http://localhost:8080/RegDone.jsp?color=Blue
Content-Type: text/html;charset=UTF-8
Content-Length: 0
Date: Sat, 28 Dec 2002 20:37:04 GMT
Server: Apache Coyote/1.0
Connection: close

LWP::Protocol::http::request: HTTP/1.1 302 Moved Temporarily
LWP::UserAgent::request: Simple response: Found
LWP::UserAgent::request: ()
LWP::UserAgent::simple_request: GET http://localhost:8080/RegDone.jsp?
color=Blue
LWP::UserAgent::_need_proxy: Not proxied
LWP::Protocol::http::request: ()
LWP::Protocol::http::request: GET /RegDone.jsp?color=Blue HTTP/1.0
Host: localhost:8080
User-Agent: libwww-perl/5.51
Cookie: JSESSIONID=887896C93DFFF372EB38818BF9F68DB2
Cookie2: $Version=1

LWP::Protocol::http::request: reading response
LWP::Protocol::http::request: HTTP/1.1 200 OK
Content-Type: text/html
Content-Length: 221
Date: Sat, 28 Dec 2002 20:37:04 GMT
```

```
Server: Apache Coyote/1.0
Connection: close

LWP::Protocol::http::request: HTTP/1.1 200 OK
LWP::Protocol::collect: read 221 bytes
LWP::UserAgent::request: Simple response: OK
```

We have a lot of information here. Let's analyze it quickly.

- My first request was sent as "GET /RegForm.jsp HTTP/1.0".
- The first response came back with a cookie as: "JSESSIONID=887896C93DFFF372EB38818BF9F68DB2". Apparently, this is the session ID, but encrypted.
- My second request was sent as "GET /RegForm.jsp?name=Mike... HTTP/1.0", with two cookies. The first cookie was the JSP server session ID. The second cookie came from nowhere.
- The second response was interesting. It had code of "302 Moved Temporarily", and a "Location" header line indicating the new URL. Obviously, this response was generated by the "sendRedirect()" function in my JSP page, RegForm.jsp.
- The LWP::UserAgent object is smart. It recognized the "302 Moved Temporarily" code, and automatically send another request with new URL location.
- With no surprises, the third response came ok. The JSP did recognize my session ID in my second and third request. But response content was missing in the debugging log for some reason. But the response was captured in the my regular output file, client.out.
- It is interesting to see that there was no cookie in the second response and third response. My guess is that JSP server saw the session ID in the second request and third request, so there was no need to put the session ID as a cookie in the responses.

Using JavaBean Classes

This chapter explains:

- How to load a JavaBean into JSP pages and manipulate its properties.
- Compilation issues of using JavaBean classes in unnamed packages.
- Setting and getting JavaBeans properties.
- Using JavaBeans as Java objects in scripting elements.
- Using Java objects as JavaBeans.
- How to refresh the JavaBean objects loaded in memory.

The "jsp:useBean" Action Elements

jsp:useBean: A JSP action element that loads a JavaBean object into the JSP page.

```
<jsp:useBean id="object_name" class="class_name"/>
```

where "object_name" is the name of the object to be created, and "class_name" is the class name of the JavaBean class from which the object will be instantiated.

Once a bean object is loaded into the page, you can use two other action elements to manipulate it.

```
<jsp:setProperty name="obj" property="prop_name" value="prop_value"/>
<jsp:getProperty name="obj" property="prop_name"/>
```

The "setProperty" action will set a new value to the specified property of the specified bean object. The "getProperty" action will get the current value of the specified property of the specified bean object. This value will be converted into a string

Once a bean object is loaded into the page, it can be used in other scripting elements in the same JSP page.

Here is my first JavaBean, CacheBean.java:

```
/**
 * CacheBean.java
 * Copyright (c) 2002 by Dr. Herong Yang. All rights reserved.
```

```
 */
public class CacheBean {
  private String text = "null";
  public String getText() {
    return text;
  }
  public void setText(String text) {
    this.text = text;
  }
  public String getInfo() {
    return "My JavaBean - Version 1.00"
  }
}
```

Here is a simple JSP page to show you how to use JavaBean, UseBean.jsp:

```
<?xml version="1.0"?>
<jsp:root xmlns:jsp="http://java.sun.com/JSP/Page" version="1.2">
<!-- UseBean.jsp
    Copyright (c) 2002 by Dr. Herong Yang
-->
<html><body>
<jsp:directive.page import="CacheBean"/>
<jsp:useBean id="b" class="CacheBean"/>
<jsp:setProperty name="b" property="text" value="Hello world!"/>
Property from my Bean:
<jsp:getProperty name="b" property="text"/>
<br/>
Info from my Bean:
<jsp:expression>b.getInfo()</jsp:expression>
</body></html>
</jsp:root>
```

Then I compiled CacheBean.java with JDK 1.3.1, and copied CacheBean.class to \local\jakarta-tomcat-4.1.18\webapps\root\web-inf\classes. Here is what I got by requesting UseBean.jsp from IE:

```
Property from my Bean: Hello world!
Info from my Bean: My JavaBean - Version 1.00
```

Note that CacheBean class needs to be imported into the JSP page. The class file needs to be copied to the .\web-inf\classes directory.

It's very interesting to see the Servlet class translated from UseBean.jsp:

```
package org.apache.jsp;

import javax.servlet.*;
import javax.servlet.http.*;
import javax.servlet.jsp.*;
import org.apache.jasper.runtime.*;
import CacheBean;

public class UseBean_jsp extends HttpJspBase {

  private static java.util.Vector _jspx_includes;

  public java.util.List getIncludes() {
    return _jspx_includes;
  }

  public void _jspService(HttpServletRequest request,
       HttpServletResponse response)
       throws java.io.IOException, ServletException {

    JspFactory _jspxFactory = null;
    javax.servlet.jsp.PageContext pageContext = null;
    HttpSession session = null;
    ServletContext application = null;
    ServletConfig config = null;
    JspWriter out = null;
    Object page = this;
    JspWriter _jspx_out = null;

    try {
      _jspxFactory = JspFactory.getDefaultFactory();
      response.setContentType("text/xml;charset=UTF-8");
      pageContext = _jspxFactory.getPageContext(this, request, response,
                  null, true, 8192, true);
      application = pageContext.getServletContext();
      config = pageContext.getServletConfig();
      session = pageContext.getSession();
      out = pageContext.getOut();
      _jspx_out = out;

      out.write("<html>");
      out.write("<body>");
      CacheBean b = null;
      synchronized (pageContext) {
```

```
      b = (CacheBean) pageContext.getAttribute("b",
          PageContext.PAGE_SCOPE);
      if (b == null){
        try {
          b = (CacheBean) java.beans.Beans.instantiate(
              this.getClass().getClassLoader(), "CacheBean");
        } catch (ClassNotFoundException exc) {
          throw new InstantiationException(exc.getMessage());
        } catch (Exception exc) {
          throw new ServletException("Cannot create bean of class "
            + "CacheBean", exc);
        }
        pageContext.setAttribute("b", b, PageContext.PAGE_SCOPE);
      }
    }
    JspRuntimeLibrary.introspecthelper(pageContext.findAttribute("b"),
      "text", "Hello world!",null, null, false);
    out.write("\nProperty from my Bean: ");
    out.print(JspRuntimeLibrary.toString(((
      (CacheBean)pageContext.findAttribute("b")).getText())));
    out.write("<br/>");
    out.write("\nInfo from my Bean: ");
    out.print(b.getInfo());
    out.write("</body>");
    out.write("</html>");
   } catch (Throwable t) {
    out = _jspx_out;
    if (out != null && out.getBufferSize() != 0)
      out.clearBuffer();
    if (pageContext != null) pageContext.handlePageException(t);
   } finally {
    if (_jspxFactory != null)
       _jspxFactory.releasePageContext(pageContext);
   }
  }
}
```

Note that:

- The "page" directive element with "import" attribute was translated into a true "import" Java statement.
- The "useBean" action element was translated into a block of code to load bean class, instantiate an object of that bean class, and put that object into the attribute collection in the page context object. The name of the attribut is set to the same as the object name.

- The "setProperty" action element was translated into a call to introspecthelper() method.
- The "getProperty" action element was translated into a call to a "get" method.
- The "expression" element was translated into a call to "out.print()".

Compilation Errors with Tomcat 4.1.18 and JDK 1.4.1

If you are using JDK 1.4.1 and trying to run the above example, you will get the following compilation error:

```
org.apache.jasper.JasperException: Unable to compile class for JSP

An error occurred at line: 8 in the jsp file: /UseBean.jsp

Generated servlet error:
    [javac] Compiling 1 source file

C:\local\jakarta-tomcat-4.1.18\work\Standalone\localhost
   \_\UseBean_jsp.java:7: '.' expected
import CacheBean;
```

This is because JDK 1.4.1 does not allow import statement to be used on classes in unnamed packages any more. For more detail information, please see section "Tomcat 4.1.18 with JDK 1.4.1" in this book.

Setting and Getting JavaBeans Properties

As you can see from previous sections, a JavaBean is just a regular Java obect. Once a JavaBean is created in a JSP, you can use JSP setProperty and getProperty action elements to set and retrieve values of its properties. To support these action elements, a JavaBean class must implement set and get methods based on some rules.

In this section, let's look at some of the basic rules about setting and getting JavaBean properties:

- The setProperty element must be supported by at least one set method in the JavaBean class.
- The set method name must be the property name with the first lower case letter being translated to upper case, and prefixed with "set". For example, "setAuthor" is a good method name to support the setProperty action element for property name "author".
- The return type of the set method should be void.
- The set method should take only one input parameter.
- If there are multiple set methods, there must be one that takes String as the input parameter type, and this will be the one to be used by the setProperty action element.
- If there is only one set method, the input parameter type should be a primitive type.

- The get method name must be the property name with the first lower case letter being translated to upper case, and prefixed with "get". For example, "getAuthor" is a good method name to support the getProperty action element for property name "author".
- The get method should take no input parameter.
- The return type of the get method can be String or any primitive type.
- JavaBean proproty names must be started with a lower case letter.

To validate the above rules, I wrote the following sample JavaBean class:

```
/**
 * DemoBean.java
 * Copyright (c) 2003 by Dr. Herong Yang. All rights reserved.
 */
package herong;
public class DemoBean {
  private String author = "Herong";
  private int count = 0;
  private boolean status = true;
  private String total = "1";
  private String size = "2";
  public void setAuthor(String a) {
    author = a;
  }
  public String getAuthor() {
    return author;
  }
  public void setCount(int c) {
    count = c;
  }
  public int getCount() {
    return count;
  }
  public void setStatus(boolean s) {
    status = s;
  }
  public boolean getStatus() {
    return status;
  }
  public void setTotal(int t) {
    total = "int: "+t;
  }
  public void setTotal(double t) {
    total = "double: "+t;
  }
```

```
  public String getTotal() {
    return total;
  }
  public void setSize(int s) {
    size = "int: "+s;
  }
  public void setSize(String s) {
    size = "String: "+s;
  }
  public String getSize() {
    return size;
  }
  public void setX(String x) {
    author = x;
  }
  public String getY() {
    return author;
  }
}
```

Compile this source code with JDK 1.4.1, and copy the class file to the Tomcat class path. Remember to store the class file under a sub directory named as "herong".

```
>\local\j2sdk1.4.1\bin\java DemoBean.java
>copy DemoBean.class
  \local\jakarta-tomcat-4.1.18\webapps\root\web-inf\classes\herong
```

Now we are ready to test this JavaBean with an JSP page, DemoBean.jsp:

```
<?xml version="1.0"?>
<jsp:root xmlns:jsp="http://java.sun.com/JSP/Page" version="1.2">
<!-- DemoBean.jsp
    Copyright (c) 2003 by Dr. Herong Yang
-->
<html><body>
<jsp:useBean id="b" class="herong.DemoBean"/>
<jsp:setProperty name="b" property="author" value="Nobody"/>
Line 1: author =
<jsp:getProperty name="b" property="author"/><br/>
<!-- <jsp:getProperty name="b" property="Author"/><br/> -->
<!-- <jsp:getProperty name="b" property="AUTHOR"/><br/> -->

<jsp:setProperty name="b" property="status" value="false"/>
Line 2: status =
```

```
<jsp:getProperty name="b" property="status"/><br/>

<jsp:setProperty name="b" property="count" value="5"/>
Line 3: count =
<jsp:getProperty name="b" property="count"/><br/>

<!-- <jsp:setProperty name="b" property="total" value="9"/> -->
Line 4: total =
<jsp:getProperty name="b" property="total"/><br/>

<jsp:setProperty name="b" property="size" value="14"/>
Line 5: size =
<jsp:getProperty name="b" property="size"/><br/>

<jsp:setProperty name="b" property="x" value="Herong"/>
Line 6: size =
<jsp:getProperty name="b" property="y"/><br/>
</body></html>
</jsp:root>
```

Make sure to run your Tomcat under JDK 1.4.1. Then open this JSP page with IE, you will get:

```
Line 1: author = Nobody
Line 2: status = false
Line 3: count = 5
Line 4: total = 1
Line 5: size = String: 14
Line 6: y = Herong
```

Note that:

- Property names are case sensitive. Property name "Author" can not be mapped to "getAuthor" method.
- Two set methods without any one taking String as input parameter type is giving me problem to set "total".
- Two set methods with one taking String as input parameter type is ok. Output line 5 is the prove.

Using JavaBeans as Objects in Scripting Elements

As I mentioned in the previous section, JavaBean is just a normal Java object with some special method. Once a JavaBean is created, we should be able to use it as Java object in any scripting elements.

Here is a sample page to show you how to use a JavaBean as a Java object. It is using the same JavaBean class, herong.DemoBean, as the previous section.

```
<?xml version="1.0"?>
<jsp:root xmlns:jsp="http://java.sun.com/JSP/Page" version="1.2">
<!-- BeanAsObject.jsp
    Copyright (c) 2003 by Dr. Herong Yang
-->
<html><body>
<jsp:useBean id="b" class="herong.DemoBean"/>
<jsp:setProperty name="b" property="author" value="Someone"/>
Line 11: author =
<jsp:expression>b.getAuthor()</jsp:expression><br/>

<jsp:scriptlet><![CDATA[b.setTotal(10);]]></jsp:scriptlet>
Line 12: total =
<jsp:getProperty name="b" property="total"/><br/>

<jsp:scriptlet><![CDATA[b.setSize(15);]]></jsp:scriptlet>
Line 13: size =
<jsp:getProperty name="b" property="size"/><br/>
Line 14: size =
<jsp:scriptlet><![CDATA[out.println(b.getSize());]]></jsp:scriptlet>
<br/>

<jsp:scriptlet><![CDATA[
  Object o = pageContext.findAttribute("b");
  String s = ((herong.DemoBean)o).getSize();
  out.println("Line 15: size = "+s);
]]></jsp:scriptlet><br/>
</body></html>
</jsp:root>
```

Open this JSP page with IE, you will get:

```
Line 11: author = Someone
Line 12: total = int: 10
Line 13: size = int: 15
Line 14: size = int: 15
Line 15: size = int: 15
```

Note that:

- Line 11 tells us that we can use an expression element to get the property value.

- Line 12 tells us that if we use scriptlet element, we can call a specific version of setTotal method. Remember setProperty failed on "total" in the previous example.
- Line 15 tells us that we also retrieve the object back from pageContext, because useBean element store the JavaBean object in pageContext.

Using Java Objects as JavaBeans

Now we know that a JavaBean is just a normal Java object with some specially named methods, and stored in pageContext's attribute collection. We can use the useBean to create a JavaBean and use it any way we wanted.

The next question is: can we create any object and put it into pageContext's attribute collection, and use setProperty and getProperty elements? The answer is yes. Here is my sample JSP page to show you this:

```
<?xml version="1.0"?>
<jsp:root xmlns:jsp="http://java.sun.com/JSP/Page" version="1.2">
<!-- ObjectAsBean.jsp
    Copyright (c) 2003 by Dr. Herong Yang
-->
<html><body>
<jsp:scriptlet><![CDATA[
  herong.DemoBean b = new herong.DemoBean();
  pageContext.setAttribute("b", b, PageContext.PAGE_SCOPE);
  b.setAuthor("Unknown");
  b.setTotal(9.99);
]]></jsp:scriptlet>
Line 21: author =
<jsp:getProperty name="b" property="author"/><br/>
Line 22: total =
<jsp:getProperty name="b" property="total"/><br/>
<jsp:scriptlet><![CDATA[b.setSize(15);]]></jsp:scriptlet>
Line 23: size =
<jsp:getProperty name="b" property="size"/><br/>
Line 24: size =
<jsp:scriptlet><![CDATA[out.println(b.getSize());]]></jsp:scriptlet>
<br/>
<jsp:scriptlet><![CDATA[
  java.util.Date d = new java.util.Date();
  pageContext.setAttribute("d", d, PageContext.PAGE_SCOPE);
]]></jsp:scriptlet>
<jsp:setProperty name="d" property="time" value="1000000000000"/>
Line 25: time =
<jsp:getProperty name="d" property="time"/><br/>
```

```
</body></html>
</jsp:root>
```

Open this JSP page with IE, you will get:

```
Line 21: author = Unknown
Line 22: total = double: 9.99
Line 23: size = int: 15
Line 24: size = int: 15
Line 25: time = 1000000000000
```

Note that:

- Line 25 tells us that we can even create a Date object and make it available a JavaBean. Its setTime and getTime methods are used provide the time property.

Refreshing the Loaded Bean Classes

Once a bean class has been used once by a JSP page, it will stay loaded in memory to avoid loading it again when another JSP page uses it. This is good to improve response time, but it is a problem if you changed your bean and wants to push the newer version into the server.

One way to force the server to use the new versions of bean classes is to shut down the server and re-start the server.

But a better way to force the server to use the new versions is to use the Tomcat Manger tool. Here is how to do this:

1. Set up a manager user name and password by adding the following line to \local\jakarta-tomcat-4.1.18\conf\tomcat-users.xml:

```
<user username="herong" password="yang" roles="manager"/>
```

2. Shut down and re-start Tomcat server.

3. Request UseBean.jsp with IE.

4. Modify CacheBean.java, compile it, and copy the class file the .\web-inf\classes.

5. Run the "reload" command with IE at http://localhost:8080/manager/reload?path=/. You need enter user name and password created in step 1.

6. Request UseBean.jsp again. You should see the changes made to CacheBean.java in step 4.

Using Cookies

What is a Cookie?

Cookie: A small amount of information sent by a Web server to a Web browser, saved by the browser, and sent back to the server later. Cookies are transmitted inside the HTTP header.

Cookies move from server to browser, and back to server as follows:

Web Server	**Web Browser**	**Local System**	**Web Browser**	**Web Server**
Send cookies	--> Receive cookies	--> Save cookies	--> Send back cookies	--> Receive cookies

As you can see from the diagram, cookies are actually saved to the hard disk of Web browser user's machines. Many users are concerned about this. But I think it is pretty safe to allow your browser to save cookies.

If you are really concerned, you can change your browser's settings to reject cookies. But this may cause many Web based applications fail to run on your browser.

Sending and Receiving Cookies

Sending cookies to the browser can be done by the addCookie() method on the response object; while receiving cookies from the browser can be done by the getCookies() method on the request object. Here is program to demonstrate how to use those methods:

```
<?xml version="1.0"?>
<jsp:root xmlns:jsp="http://java.sun.com/JSP/Page" version="1.2">
<!--
 - CookieTest.jsp
 - Copyright (c) 2002 by Dr. Herong Yang. All rights reserved.
-->
<jsp:directive.page contentType="text/html"/>
<html><body>
<p>
<jsp:directive.page import="javax.servlet.http.Cookie"/>
```

```
<jsp:scriptlet><![CDATA[
  out.println("<b>Cookies received by the server:</b><br/>");
  Cookie[] cookies = request.getCookies();
  int n = 0;
  if (cookies!=null) {
    n = cookies.length;
    for (int i=0; i<cookies.length; i++) {
      out.println(cookies[i].getName()+": "
        +cookies[i].getValue()+"<br/>");
    }
  }
  out.println("<b>Cookies added by the server:</b><br/>");
  Cookie c = new Cookie("Cookie_"+n,"value");
  out.println(c.getName()+": "+c.getValue()+"<br/>");
  response.addCookie(c);
]]></jsp:scriptlet>
</p>
</body></html>
</jsp:root>
```

So I opened this page with IE, I got:

```
Cookies received by the server:
Cookies added by the server:
Cookie_0: value
```

Then I clicked the refresh button on the IE window, I got:

```
Cookies received by the server:
Cookie_0: value
JSESSIONID: 7E33A51C5F05A11647467E1735C5084E
Cookies added by the server:
Cookie_2: value
```

What happened here was that when I opened the page the first time, the server received no cookie from the browser's request. But my program added one cookie named as "Cookie_0" to the response, and the JSP server also added a cookie named as "JSESSIONID".

When I clicked the refresh button, the browser sent the two cookies back to the server in the request. Then my program added another cookie named as "Cookie_2" in the response.

If I keep clicking the refresh button, more and more cookies would be added to the request and response. But there is a limit. The browser will only take up to 20 cookies from one Web server.

Persistent Cookies

There are two kinds of cookies: persistent cookies and temporary cookies.

A persistent cookie is one stored as a file on your computer, and it remains there when you close Internet Explorer. The cookie can be read by the Web site that created it when you visit that site again. You can use cookie.setMaxAge(s) to set the cookie to be persistent for 's' seconds.

A temporary or session cookie is stored only for your current browsing session, and is deleted from your computer when you close Internet Explorer. You can use cookie.setMaxAge(-1) to set the cookie to be temporary.

Other properties of a cookie object that you can set and get:

- setPath(): Defines the path name so that when a pages under this path name is requested, the browser will send this cookie to the server. The path name is the url without the server name and the port number parts.
- setDomain(): Defines the domain name so that when a page under this domain name is requested, the browser will send this cookie to the server. The default domain name is the server name from which the cookie was created.
- setMaxAge(): Defines how long the cookie should be stored on the browser's machine. MaxAge value is in unit of second. A negative value tells the browser to not save it to the harddisk. A zero value tells the browser to delete it.
- setVersion(): Defines the version number to which this cookie is compliant with.

The following JSP page shows you how to set a persistent cookie and how to use other properties.

```
<?xml version="1.0"?>
<jsp:root xmlns:jsp="http://java.sun.com/JSP/Page" version="1.2">
<!--
 - CookieProperties.jsp
 - Copyright (c) 2003 by Dr. Herong Yang. All rights reserved.
-->
<jsp:directive.page contentType="text/html"/>
<html><body>
<p>
<jsp:directive.page import="javax.servlet.http.Cookie"/>
<jsp:scriptlet><![CDATA[
// Setting a cookie with default properties
   out.println("<b>Cookie with default properties:</b><br/>");
   Cookie c = new Cookie("Date","30-Mar-2003");
   response.addCookie(c);
```

```
   out.println("Name: "+c.getName()+"<br/>");
   out.println("Value: "+c.getValue()+"<br/>");
   out.println("Domain: "+c.getDomain()+"<br/>");
   out.println("Path: "+c.getPath()+"<br/>");
   out.println("MaxAge: "+c.getMaxAge()+"<br/>");
   out.println("Version: "+c.getVersion()+"<br/>");

 // Setting a cookie with specified properties
   out.println("<b>Cookie with specified properties:</b><br/>");
   c = new Cookie("User","Herong Yang");
   c.setMaxAge(3*24*60*60);
   response.addCookie(c);
   out.println("Name: "+c.getName()+"<br/>");
   out.println("Value: "+c.getValue()+"<br/>");
   out.println("Domain: "+c.getDomain()+"<br/>");
   out.println("Path: "+c.getPath()+"<br/>");
   out.println("MaxAge: "+c.getMaxAge()+"<br/>");
   out.println("Version: "+c.getVersion()+"<br/>");

 // Checking properties of the received cookies
   out.println("<b>Properties of the received cookies:</b><br/>");
   Cookie[] cookies = request.getCookies();
   int n = 0;
   if (cookies!=null) {
     n = cookies.length;
     for (int i=0; i<cookies.length; i++) {
       out.println("Name: "+cookies[i].getName()+"<br/>");
       out.println("Value: "+cookies[i].getValue()+"<br/>");
       out.println("Domain: "+cookies[i].getDomain()+"<br/>");
       out.println("Path: "+cookies[i].getPath()+"<br/>");
       out.println("MaxAge: "+cookies[i].getMaxAge()+"<br/>");
       out.println("Version: "+cookies[i].getVersion()+"<br/>");
     }
   }
 ]]></jsp:scriptlet>
 </p>
 </body></html>
 </jsp:root>
```

So I opened this page with IE, and got:

```
 Cookie with default properties:
 Name: Date
 Value: 30-Mar-2003
```

Domain: null
Path: null
MaxAge: -1
Version: 0
Cookie with specified properties:
Name: User
Value: Herong Yang
Domain: null
Path: null
MaxAge: 259200
Version: 0
Properties of the received cookies:

Then I clicked at IE "Tools" menu, selected "Internet Options...". and clicked the "Settings..." button in the "Temporary Internet files" section of the "General" tab. I saw where is my "Temporary Internet files folder". So I went to that folder, and saw a cookie file named something like "Cookie:user@localhost/jsp/". I double clicked on that file, and was able to open it in notepad:

User
Herong Yang
localhost/jsp/
1024
2353942784
29567146
224352272
29566543
*

Then I clicked the IE refresh button, I got the following in the IE window:

Cookie with default properties:
Name: Date
Value: 30-Mar-2003
Domain: null
Path: null
MaxAge: -1
Version: 0
Cookie with specified properties:
Name: User
Value: Herong Yang
Domain: null
Path: null
MaxAge: 259200

Version: 0
Properties of the received cookies:
Name: Date
Value: 30-Mar-2003
Domain: null
Path: null
MaxAge: -1
Version: 0
Name: User
Value: Herong Yang
Domain: null
Path: null
MaxAge: -1
Version: 0
Name: JSESSIONID
Value: 37CB87D855A94F84355FE86626D2BAF7
Domain: null
Path: null
MaxAge: -1
Version: 0

It is interesting to know that:

- The setMaxAge() did force the Web browser to save the cookie to the local hard disk.
- The persisted cookie "User" has the default domain "localhost" and default path "/jsp".
- When this persisted cookie "User" was posted to the server again, the MaxAge value got reset to "-1" again.
- Cookie "SESSIONID" was added by the Web server, not by my program.

Controlling HTTP Response Header Lines

HTTP Response Syntax

Based on HTTP/1.1 protocol, after receiving and interpreting an HTTP request from a client, a server must responds with an HTTP response following the syntax bellow:

```
status-line
header-line
...
header-line

entity-body
```

Note that:

- Response must have one status-line.
- Response can have zero, one, or many header lines.
- Response can only have zero or one entity-body.
- There is a blank line between header lines and the entity body.
- Status line, head line, and blank line must be ended with CRLF ("/r/n") characters.
- Entity body is the actual data requested by the client request.
- Header lines can be in any order.

Bellow is a simple HTTP response with two header lines:

```
HTTP/1.1 200 OK
Content-Type: text/html
Content-Length: 38

Hello world!
```

HTTP Response Header Lines

HTTP/1.1 response header lines allows the server to passes additional information about the response which cannot be placed in the status line. Header lines can be divided into three groups:

1. General header lines: Information about the transmission of the entire response message:

   Cache-Control
   Connection
   Date
   Pragma
   Trailer
   Transfer-Encoding
   Upgrade
   Via
   Warning

2. Response header lines: Information about the response:

   Accept-Ranges
   Age
   ETag
   Location
   Proxy-Authenticate
   Retry-After
   Server
   Vary
   WWW-Authenticate

3. Entity header lines: Information about the data requested by the client:

   Allow
   Content-Encoding
   Content-Language
   Content-Length
   Content-Location
   Content-MD5
   Content-Range
   Content-Type
   Expires
   Last-Modified

Controlling Response Header Lines

When a JSP page is requested, the response header lines will be created by the JSP server. But you can indirectly control some header lines in three 3 different ways:

1. Using a directive element to set the entity header line: Content_Type, as shown in the

following example:

```
<jsp:directive.page contentType="text/html"/>
```

2. Using special methods on the "response" object, as defined by the javax.servlet.ServletResponse interface, to set the entity header lines: Content_Type and Content_Length as shown in the following example:

```
response.setContentType("text/html");
responee.setContentLength(909);
```

3. Using generic methods on the "response" object, as defined by the javax.servlet.http.HttpServletResponse interface, to add or set any response header lines as shown in the following example:

```
response.setHeader("Content_Type", "text/html");
response.setIntHeader("Content_Length", 909);
response.setDateHeader("Last-Modified", System.currentTimeMillis());
response.addHeader("Content_Type", "charset=ISO-8859-1");
```

Viewing Response Header Lines

When the client program receives the HTTP response, it will look at the header lines first. Based on the information contained in the header lines, the client program will decide what to do with the actual response data in the entity body.

If you use a Web browser as a HTTP client program, it will process the data in the entity body differently depending on mainly the "Content_Type" entity header line: displaying the data as it is, rendering the data as a HTML document and displaying the resulting information, or passing the data to other registered programs to handle it.

Once the Web browser finishes processing the entity body, you can get some limited information from the header lines. For example, you can click the right mouse button and select the properties command on Internet Explorer, it will display some general properties about this response in a pop up window. The properties displayed are not always identical to the response header lines. The "Modified" property is probably identical to the "Last_Modified" entity header line. The "Type" property is sometime related to the "Content_Type" entity header line, and sometimes related to server side resource that generated the response. For example, if you use Internet Explorer to request hello.jsp from a JSP Web server, and view the page properties, you will see "JavaServer Page" in the "Type" property. But the "Content_Type" header line received from this JSP page is "text/html".

How to view all the header lines received in the HTTP response? I couldn't find any existing tools to do this. So wrote the following program to dump the entire response including all

header lines received from a Web server:

```
/**
 * HttpRequestGet.java
 * Copyright (c) 2002 by Dr. Herong Yang. All rights reserved.
 */
import java.io.*;
import java.net.*;
public class HttpRequestGet {
  public static void main(String[] args) {
    String path = "/index.html";
    int port = 80;
    String host = "localhost";
    if (args.length > 0) path = args[0];
    if (args.length > 1) port
      = Integer.valueOf(args[1]).intValue();
    if (args.length > 2) host = args[2];
    String result = "";
    try {
      Socket c = new Socket(host,port);
      BufferedWriter w = new BufferedWriter(new OutputStreamWriter(
        c.getOutputStream()));
      BufferedReader r = new BufferedReader(new InputStreamReader(
        c.getInputStream()));
      String m = "GET "+ path + " HTTP/1.0";
      w.write(m,0,m.length());
      w.newLine();
      w.newLine();
      w.flush();
      while ((m=r.readLine())!= null) {
        System.out.println(m);
      }
      w.close();
      r.close();
      c.close();
    } catch (IOException e) {
      System.err.println(e.toString());
    }
  }
}
```

Response Header Lines of Static Files

Static files can be served directly by Tomcat server, if you copy the files to

\local\jakarta-tomcat-4.1.18\webapps\ROOT. Tomcat server will set "Content_Type" header line based on the file name extension and the MIME settings of the server configuration. Let's look at 3 commonly used file name extensions.

1. Command: "java HttpRequestGet /hello.html 8080" gives us:

```
HTTP/1.1 200 OK
ETag: W/"38-1047477954000"
Last-Modified: Sat, 22 Mar 2003 14:05:54 GMT
Content-Type: text/html
Content-Length: 38
Date: Sun, 23 Mar 2003 02:59:32 GMT
Server: Apache Coyote/1.0
Connection: close

Hello world!
```

Couple of interesting notes here:

- Content-Type was set to "text/html", because the file name extension was "html".
- The request was marked as HTTP/1.0 in HttpRequestGet, but Tomcat responded with a higher version, HTTP/1.1.
- I also tried to use HTTP/1.1 in my request, but Tomcat returned with an error. Why Tomcat could not support HTTP/1.1 request?

2. Command: "java HttpRequestGet /dot.gif 8080" gives us:

```
HTTP/1.1 200 OK
ETag: W/"43-1029361700000"
Last-Modified: Sun, 11 Aug 2002 21:48:20 GMT
Content-Type: image/gif
Content-Length: 43
Date: Sun, 23 Mar 2003 03:14:22 GMT
Server: Apache Coyote/1.0
Connection: close

GIF89a.....
```

As you can see, Content_Type was set correctly to "image/gif" for file name extension "gif", as defined in the MIME settings. I could not included the entity body here because it contains binary data.

3. Command: "java HttpRequestGet /hello.pdf 8080" gives us:

```
HTTP/1.1 200 OK
```

```
ETag: W/"909-1059340932000"
Last-Modified: Sun, 17 Mar 2003 21:22:12 GMT
Content-Type: application/pdf
Content-Length: 909
Date: Sat, 23 Mar 2003 03:29:21 GMT
Server: Apache Coyote/1.0
Connection: close

%PDF-1.3
% ...
4 0 obj
......
```

Again, Content_Type was set correctly to "application/pdf" for file name extension "pdf", as defined in the MIME settings. I truncated the entity body to save some space.

Response Header Lines Affected by jsp:directive.page Elements

As I mentioned earlier, the first way to control the response header lines is to use "jsp:directive.page" elements. Let me use the following 3 example JSP pages to show you how to do this.

Copy the first example JSP page, hello.jsp, to Tomcat server:

```
<html><body>
<% out.println("Hello world!"); %>
</body></html>
```

Then obtain the response with "java HttpRequestGet /hello.jsp 8080":

```
HTTP/1.1 200 OK
Set-Cookie: JSESSIONID=4BEF55D47FC7A80A75A97082756B772E; Path=/
Content-Type: text/html;charset=ISO-8859-1
Content-Length: 44
Date: Sat, 23 Mar 2003 21:48:54 GMT
Server: Apache Coyote/1.0
Connection: close

<html><body>
Hello world!

</body></html>
```

hello.jsp was written in an HTML format with embedded JSP statements, so Tomcat decided to

set Content_Type to "text/html;charset=ISO-8859-1", which is perfectly ok.

Copy the second example JSP page, hello_xml.jsp, Tomcat server:

```
<?xml version="1.0"?>
<jsp:root xmlns:jsp="http://java.sun.com/JSP/Page" version="1.2">
<!-- hello_xml.jsp
    Copyright (c) 2002 by Dr. Herong Yang
-->
<html><body>
<jsp:scriptlet>out.println("Hello world!");</jsp:scriptlet>
</body></html>
</jsp:root>
```

Then obtain the response with "java HttpRequestGet /hello_xml.jsp 8080":

```
HTTP/1.1 200 OK
Set-Cookie: JSESSIONID=94F75D820DA1B0BA6023704C5D2E665C; Path=/
Content-Type: text/xml;charset=UTF-8
Content-Length: 40
Date: Sat, 23 Mar 2003 21:59:03 GMT
Server: Apache Coyote/1.0
Connection: close

<html><body>Hello world!
</body></html>
```

This time, hello_xml.jsp was written in XML format, so Tomcat decided to set Content_Type to "text/xml;charset=UTF-8". This is not right, because it doesn't match the entity body. If you use Internet Explorer to request this JSP page, the entity body will not be rendered as XML data.

In the third example JSP page, hello_xml_html.jsp, I used the jsp:directive.page element to correct the problem in the second example:

```
<?xml version="1.0"?>
<jsp:root xmlns:jsp="http://java.sun.com/JSP/Page" version="1.2">
<!-- hello_xml_html.jsp
    Copyright (c) 2002 by Dr. Herong Yang
-->
<jsp:directive.page contentType="text/html"/>
<html><body>
<jsp:scriptlet>out.println("Hello world!");</jsp:scriptlet>
</body></html>
```

```
</jsp:root>
```

Obtain the response with "java HttpRequestGet /hello_xml_html.jsp 8080":

```
HTTP/1.1 200 OK
Set-Cookie: JSESSIONID=4B6E955411EBC4536206978E3B498B50; Path=/
Content-Type: text/html;charset=UTF-8
Content-Length: 40
Date: Sat, 23 Mar 2003 22:11:40 GMT
Server: Apache Coyote/1.0
Connection: close

<html><body>Hello world!
</body></html>
```

As you can see from the response, the attribute "contentType" in the "jsp:directive.page" changed the "Content_Type" header line. Note that the "charset" portion was not changed, because no value was given in the "contentType" attribute.

Setting Header Lines Directly in JSP Pages

The second way and third way to control the response header lines are related to the build-in response object. One is to use the specialized methods, the other is to use the generic methods. Let me use the following 2 examples to show you how those methods work.

The first example, SetContentType.jsp, uses the special methods to set Content_Type and Content_Length header lines:

```
<?xml version="1.0"?>
<jsp:root xmlns:jsp="http://java.sun.com/JSP/Page" version="1.2">
<!-- SetContentType.jsp
    Copyright (c) 2002 by Dr. Herong Yang
-->
 <jsp:scriptlet><![CDATA[
  response.setContentType("text/html");
  String text = "<html><body>Hello world!</body></html>";
  response.setContentLength(text.length());
  out.print(text);
 ]]></jsp:scriptlet>
</jsp:root>
```

The response:

```
HTTP/1.1 200 OK
Set-Cookie: JSESSIONID=1771F49508924D74AA8B29BB9AB770C8; Path=/
Content-Type: text/html
Content-Length: 38
Date: Tue, 12 Aug 2003 23:05:34 GMT
Server: Apache Coyote/1.0
Connection: close

<html><body>Hello world!</body></html>
```

Note that the setContentType() method overrides the Content_Type header line completely, including the charset portion. Here is another calling example, response.setContentType("text/html;charset=UTF-8").

In the second example, SetHeader.jsp, I was trying to use the generic methods to set various header lines:

```
<?xml version="1.0"?>
<jsp:root xmlns:jsp="http://java.sun.com/JSP/Page" version="1.2">
<!-- SetHeader.jsp
    Copyright (c) 2002 by Dr. Herong Yang
-->
 <jsp:scriptlet><![CDATA[
  response.setHeader("Content_Type","text/html");
  response.setIntHeader("Content_Length",0);
  String text = "<html><body>Hello world!</body></html>";
  response.setIntHeader("Content_Length",text.length());
  response.setDateHeader("Last-Modified", System.currentTimeMillis());
  response.setHeader("Author", "Herong Yang");
  response.addHeader("Author", "Joe Wang");
  out.print(text);
 ]]></jsp:scriptlet>
</jsp:root>
```

Here is the response:

```
HTTP/1.1 200 OK
Set-Cookie: JSESSIONID=6C081157BFD264C3222FB71728C00B4C; Path=/
Content_Type: text/html
Content_Length: 38
Last-Modified: Sat, 23 Mar 2003 13:46:53 GMT
Author: Herong Yang
Author: Joe Wang
Content-Type: text/xml;charset=UTF-8
```

```
Content-Length: 38
Date: Sat, 23 Mar 2003 13:46:53 GMT
Server: Apache Coyote/1.0
Connection: close

<html><body>Hello world!</body></html>
```

Note that:

- There are two "Content_Type" header lines, one from my setHeader() call, and one added by Tomcat. My guess is that Tomcat does not recognize the Content_Type header line generated by the addHeader() method. So we have to use setContentType() method to control the Content_Type header line.
- The same issue also exists on the "Content_Length" header line. We have to use setContentLength() method to control the Content_Length header line.
- The setIntHeader() method is called twice with the same header line name "Content_Length". The second call overrides the first call.
- I added two new header lines called "Author".

Here is the revised version of the second example, SetHeaderRevised.jsp:

```
<?xml version="1.0"?>
<jsp:root xmlns:jsp="http://java.sun.com/JSP/Page" version="1.2">
<!-- SetHeaderRevised.jsp
    Copyright (c) 2002 by Dr. Herong Yang
-->
 <jsp:scriptlet><![CDATA[
  response.setContentType("text/html;charset=UTF-8");
  String text = "<html><body>Hello world!</body></html>";
  response.setContentLength(text.length());
  response.setDateHeader("Last-Modified", System.currentTimeMillis());
  response.setHeader("Author", "Herong Yang");
  out.print(text);
 ]]></jsp:scriptlet>
</jsp:root>
```

Generating Non-HTML Entity Body

Sometimes, you may want to send back information in the entity body that are not in the HTML format, for example, a PDF document, or MS Word Document. In this case, we have to set Content_Type, Content_Length and other header lines carefully to provide correct information about the entity body for the client program. Here is a sample JSP page to show you how to set header lines for different types of data in the entity body.

```
<?xml version="1.0"?>
<jsp:root xmlns:jsp="http://java.sun.com/JSP/Page" version="1.2">
<!-- GetFile.jsp
    Copyright (c) 2002 by Dr. Herong Yang
-->
<jsp:directive.page session="false" import="java.io.*" />
<jsp:scriptlet>
  String p = request.getQueryString();
  boolean ok = true;
  ok = p!=null;
  if (ok) {
    if (p.indexOf(".html")>-1) {
      response.setContentType("text/html");
    } else if (p.indexOf(".gif")>-1) {
      response.setContentType("image/gif");
    } else if (p.indexOf(".pdf")>-1) {
      response.setContentType("application/pdf");
    } else if (p.indexOf(".doc")>-1) {
      response.setContentType("application/msword");
    } else {
      ok = false;
    }
  }
  if (ok) {
    try {
      int l = (int) new File(p).length();
      response.setContentLength(l);
      byte[] b = new byte[l];
      FileInputStream f = new FileInputStream(p);
      f.read(b);
      ServletOutputStream o = response.getOutputStream();
      o.write(b,0,l);
      o.flush();
      o.close();
      f.close();
    } catch (Exception e) {
      ok = false;
    }
  }
  if (!ok) {
    response.sendError(HttpServletResponse.SC_BAD_REQUEST);
  }
 </jsp:scriptlet>
</jsp:root>
```

Ideas used in this page:

- The objective of this page is to send back the content of the requested file in entity body, and set the Content_Type and Content_Length header lines correctly.
- The requested file name is given in the query string of the HTTP request.
- The extension of the requested file name is checked to determine the Content_Type header line.
- Then the requested file size is checked to set the Content_Length header line.
- Then the requested file is opened, and the content is copied to output stream of the response object as a byte array.
- If any thing goes wrong, an error status code is send to the response.

Now let's see how this page works.

1. Use IE (Internet Explorer) to request: http://localhost:8080/GetFile.jsp?hello.html, you should see the hello message properly displayed as HTML document.

2. Use IE to request: http://localhost:8080/GetFile.jsp?dot.gif, you should see a tiny dot displayed as an image.

3. Use IE to request: http://localhost:8080/GetFile.jsp?hello.pdf, you should see IE calling Adobe Reader to display the hello message as a PDF document.

4. Use IE to request: http://localhost:8080/GetFile.jsp?hello.doc, you should see IE calling MS Word to display the hello message as Word document. Of course, you have prepare such a Word document and put it on Tomcat server in order to do this test.

5. Use IE to request: http://localhost:8080/GetFile.jsp?any.txt, you should see IE displaying an error message. The reason is, of course, that the requested file doesn't exist.

Another way of sending non-HTML data to the client is via attachment. The following JSP will show you how to do this:

```
<?xml version="1.0"?>
<jsp:root xmlns:jsp="http://java.sun.com/JSP/Page" version="1.2">
<!-- Download.jsp
    Copyright (c) 2002 by Dr. Herong Yang
-->
<jsp:directive.page session="false" import="java.io.*" />
<jsp:scriptlet>
  String p = request.getQueryString();
  boolean ok = true;
  ok = p!=null;
  if (ok) {
    if (p.indexOf(".html")>-1) {
```

```
      response.setContentType("text/html");
    } else if (p.indexOf(".gif")>-1) {
      response.setContentType("image/gif");
    } else if (p.indexOf(".pdf")>-1) {
      response.setContentType("application/pdf");
    } else if (p.indexOf(".doc")>-1) {
      response.setContentType("application/msword");
    } else {
      ok = false;
    }
  }
  if (ok) {
    response.setHeader("Content-disposition",
      "attachment; filename="+p);
    try {
      int l = (int) new File(p).length();
      response.setContentLength(l);
      byte[] b = new byte[l];
      FileInputStream f = new FileInputStream(p);
      f.read(b);
      ServletOutputStream o = response.getOutputStream();
      o.write(b,0,l);
      o.flush();
      o.close();
      f.close();
    } catch (Exception e) {
      ok = false;
    }
  }
  if (!ok) {
    response.sendError(HttpServletResponse.SC_BAD_REQUEST);
  }
 </jsp:scriptlet>
</jsp:root>
```

In this page, anther header line, "Content-disposition", is added to the response, in which I am telling the client program that the entity data is an attachment, with file name specified.

Now try to use IE to request: http://localhost:8080/Download.jsp?hello.pdf, you will see IE prompting you to save the attachment instead of calling Adobe Reader to display the data.

IE 6.0 Bug on Display PDF Data

While I was trying to display PDF data, I found that IE 6.0 is not responding correctly to

Content-Type: application/pdf. Here is a sample JSP, GetPdf.jsp, to show the problem.

```
<?xml version="1.0"?>
<jsp:root xmlns:jsp="http://java.sun.com/JSP/Page" version="1.2">
<!-- GetPdf.jsp
    Copyright (c) 2002 by Dr. Herong Yang
-->
<jsp:directive.page session="false" import="java.io.*" />
<jsp:scriptlet>
  String p = "hello.pdf";
  response.setContentType("application/pdf");
  try {
    int l = (int) new File(p).length();
    response.setContentLength(l);
    byte[] b = new byte[l];
    FileInputStream f = new FileInputStream(p);
    f.read(b);
    ServletOutputStream o = response.getOutputStream();
    o.write(b,0,l);
    o.flush();
    o.close();
    f.close();
  } catch (Exception e) {
    response.sendError(HttpServletResponse.SC_BAD_REQUEST);
  }
 </jsp:scriptlet>
</jsp:root>
```

If you use IE 6.0 to request: http://localhost:8080/GetPdf.jsp, you will get nothing on the IE window.

Now if you use IE 6.0 to request: http://localhost:8080/GetPdf.jsp?x.pdf, you will see Adobe Reader displaying the hello message.

Interestingly, if you use IE 6.0 to request: http://localhost:8080/GetPdf.jsp?x.doc, you will also see Adobe Reader displaying the hello message.

I guess IE 6.0 has a stupid bug. What do you think?

Localization / Internationalization - Non ASCII Characters in JSP Pages

This chapter explains:

- How characters travel from JSP files to browser screens.
- How ASCII characters work in JSP pages.
- How to present non ASCII characters in HTML documents.
- How to enter non ASCII charactetrs in Java strings and contole the output encodings.
- How Java string works with byte sequences encoded for a local language.
- How Java string works with Unicode codes - local language independent.
- How to enter non ASCII characters as static HTML text.
- How static HTML text works in HTML pages
- How static HTML text works in JSP pages with standard syntax.
- How static HTML text works in JSP pages with XML syntax.
- How to supporting characters from multiple languages.

For more notes on non ASCII codes and Java program localization, see my other books: "Herong's Notes on Unicode" and "Herong's Notes on JDK".

Characters Traveling from JSP Files to Browser Screens

Handling non ASCII characters in JSP files correctly is not an easy task. I have seen many messages on the Wep in this area reporting various frustrating situations. One main reason is that text entered in a JSP file must travel through many steps before being displayed by a browser on a screen.

The following diagram illustrates steps that characters must travel from a JSP file to a browser screen, and computing technologies that are used at different steps:

0. Key Sequences from Keyboard
   |
   |- Text Editor
   v
1. JSP File
   |
   |- XML Parser
   v

2. Java Source File
   |
   |- Java Compiler
   v
3. Java Class File
   |
   |- Java Virtual Machine
   v
4. HTML Document
   |
   |- Web Server
   v
5. HTTP Response
   |
   |- Internet TCP/IP Connection
   v
6. HTTP Response
   |
   |- Web Browser
   v
7. Characters on the Screen

Since all computing technologies are using ASCII encoding by default, text of ASCII characters can safely travel through those steps without any issues.

However, for non ASCII characters, we have to watch out each steps carefully to make sure that characters are not damaged, and/or decoded correctly if encoded.

ASCII Characters in JSP Pages

As I mentioned earlier, ASCII characters can travel from JSP files to browsers easily without any trouble. Here is a simple JSP file with ASCII characters only:

```
<?xml version="1.0"?>
<jsp:root xmlns:jsp="http://java.sun.com/JSP/Page"
  xmlns:c="http://java.sun.com/jstl/core" version="1.2">
<!-- HelpASCII.jsp
   Copyright (c) 2002 by Dr. Herong Yang
-->
<jsp:directive.page contentType="text/html"/>
<html><body>
<b>Help</b><br/>
<p>This is a very simple help page...</p>
```

```
</body></html>
</jsp:root>
```

If you view this page with a browser, you will get two lines of characters:

```
Help
This is a very simple help page...
```

They are exactly what I entered into the JSP file.

Presenting Non ASCII Characters in HTML Documents

In order to ensure non ASCII characters entered in JSP files to show up on browser screens correctly, we need to understand how non ASCII characters are processed from one step to the other. The processing steps can be grouped into two parts:

- Outputing HTML documents with non ASCII characters - steps 0, 1, 2, and 3.
- Presenting non ASCII characters in HTML documents - steps 4, 5, 6, and 7.

Let's look at the second part first to see how non ASCII characters are stored in HTML documents, transferred from Web servers to browsers, displayed on the screen. Here are some basic rules related to these steps:

- Non ASCII characters must be encoded in a particular encoding schema, like GB2312, Shift-JIS, or UTF-8.
- You only use a single encoding schema in one HTML document.
- The encoding schema name should be given in a meta tag as the charset value. For examples, see my sample HTML document in this section.
- Non ASCII characters can be transferred safely from Web servers to browsers.
- The browser must decode HTML documents based on the schema name given in the documents - auto mode, or set by the browser user - manual mode.
- Once non ASCII characters are decoded correctly, the browser must be provided with font files that match the character set in which those non ASCII characters are defined.

In order to test these rules, I translated my HelpASCII.html to Chinese with GB2312 encoding schema, and saved in a file called, HelpGB2312.html:

```
<html>
<!-- HelpGB2312.html
   Copyright (c) 2002 by Dr. Herong Yang
-->
<meta http-equiv="Content-Type" content="text/html; charset=gb2312"/>
<body>
<b>??</b><br/>
```

```
<p>????????????…</p>
</body>
</html>
```

You may have trouble read this file on this page, or copy it to your local system, because it contains non ASCII characters. Bellow is the same file in hex number format. You can use it to fix or regenerate HelpGB2312.html.

```
3C68746D6C3E0D0A3C212D2D2048656C
704742323331322E68746D6C0D0A2020
202020436F7079726967687420286329
203230303420627920447265616D7765
6F6E672059616E670D0A2D2D3E0D0A3C
6D65746120687474702D65717569763D
22436F6E74656E742D54797065222063
6F6E74656E743D22746578742F68746D
6C3B20636861727365743D6762323331
32223E0D0A3C626F64793E0D0A3C623E
CBB5C3F73C2F623E3C62722F3E0D0A3C
703ED5E2CAC7D2BBB7DDB7C7B3A3BCE4
B5A5B5C4CBB5C3F7CAE9A1AD3C2F703E
0D0A3C2F626F64793E0D0A3C2F68746D
6C3E0D0A
```

When I opened HelpGB2312.html with IE (Internet Explorer), I saw Chinese characters correctly displayed on the screen. I verified my IE encoding settings, View menu and Encoding command, it has "Auto-select" checked, and Chinese Simplified (GB2312) selected. I also verified my IE font settings, Tools menu, Internet Options command, and Fonts button, it has fonts installed for Chinese Simplified language.

When I changed my IE encoding setting to another encoding, like UTF-8, I got strange characters showing up on the screen, because I forced IE to decode my GB2312 encoded document with UTF-8 encoding schema.

Entering Non ASCII Characters in Java Strings

Now let's look at the first part of the process to see how non ASCII chararters can be entered in JSP pages, converted into Java programs, and outputed into HTML documents. Rules related to these steps are:

- Non ASCII characters can be entered JSP pages in two ways: as static HTML text, and as dynamic Java statements.
- Java strings are sequences of 2-byte characters.
- Non ASCII characters can be entered into Java string literals as Unicode codes in \u hex

digits sequences.
- Non ASCII characters can also be entered into Java string literals as Unicode codes in UTF-8 encoding byt sequences. You may need a UTF-8 sensitive editor to enter your Java source code, because regular text editor may not be able to recongnize UTF-8 byte sequences.
- Java can convert Unicode codes to various local language codes as encoding processes at the character based output stream level.
- JSP server object "response" offers two output streams: response.getWriter(), and response.getOutputStream(). You can only use one of the two streams in a single JSP page.
- response.getWrite() allows you to output characters with Unicode encoding specified by the response.setContentType() method.
- response.getOutputStream() allows you to output binary bytes.
- Static HTML text will be converted into out.write() statemenss
- JSP page can be written as an XML file, which requires XML encoding rules.

Based these rules, we have three options to output a HTML document with non ASCII characters:

- 1. Enter non ASCII characters in the encoded form required by the HTML document as sequence of types, and use Java binary output stream to generate the HTML document.
- 2. Enter non ASCII characters in Unicode codes, and use Java writer output stream to generate the HTML with the stream set to the encoding required by the HTML document.
- 3. Enter non ASCII characters as static HTML text, and let the JSP server to convert them into out.write() statements to generate the HTML document.

Java Strings - Byte Sequences Encoded for Local Languages

Let's try option 1 mentioned in the previous section first. Here is my sample JSP page:

```
<?xml version="1.0"?>
<jsp:root xmlns:jsp="http://java.sun.com/JSP/Page"
  xmlns:c="http://java.sun.com/jstl/core" version="1.2">
<!-- HelpGB2312Java.jsp
    Copyright (c) 2002 by Dr. Herong Yang
-->
<jsp:directive.page contentType="text/html; charset=gb2312"/>
<jsp:declaration><![CDATA[
  private java.io.OutputStream outStream;
  private void writeGB(String s) throws Throwable {
    for (int i=0; i<s.length(); i++) {
      char c = s.charAt(i);
      byte b = (byte) (c>>8 & 0x00FF);
```

```
        //if (b>0)
        outStream.write(b);
        b = (byte) (c & 0x00FF);
        outStream.write(b);
      }
    }
 ]]></jsp:declaration>
 <jsp:scriptlet><![CDATA[
    outStream = response.getOutputStream();
    writeGB("<html>");
    writeGB("<meta http-equiv=\"Content-Type\""
      + " content=\"text/html; charset=gb2312\"/>");
    writeGB("<body>");
    writeGB("<b>\uCBB5\uC3F7</b>");
    writeGB("<p>\uD5E2\uCAC7\uD2BB\uB7DD\uB7C7\uB3A3\uBCE4\uB5A5"
      + "\uB5C4\uCBB5\uC3F7\uCAE9\uA1AD</p>");
    writeGB("</body>");
    writeGB("</html>");
 ]]></jsp:scriptlet>
 </jsp:root>
```

When I opened HelpGB2312Java.jsp with IE, I saw Chinese characters correctly displayed on the screen. So option 1 works! But note that:

- Figuring out the byte sequences of non ASCII characters in a particular encoding is not that hard. Simplified Chinese text files are usually written in byte sequences of GB2312 encoding.
- Byte sequences can only be entered in Java statements in Hex number format.
- response.getOutputStream() need to be called before any other output statements.
- Once response.getOutputStream() is called, you can not call response.getWriter() any more. So the entire HTML document must be outputed in binary mode.
- You can not add any static HTML text, because that requires response.getWriter().
- A JSP directive.page element is needed to set the Content-Type header of the HTML reponse with the sample charset value as the HTML document.

Java Strings - Unicode Codes - Local Language Independent

Let's try option 2 now. Here is my sample JSP page:

```
 <?xml version="1.0"?>
 <jsp:root xmlns:jsp="http://java.sun.com/JSP/Page"
    xmlns:c="http://java.sun.com/jstl/core" version="1.2">
 <!-- HelpGB2312Unicode.jsp
```

```
    Copyright (c) 2002 by Dr. Herong Yang
-->
<jsp:scriptlet><![CDATA[
  response.setContentType("text/html; charset=gb2312");
  out.println("<html>");
  out.println("<meta http-equiv=\"Content-Type\""
    + " content=\"text/html; charset=gb2312\"/>");
  out.println("<body>");
  out.println("<b>\u8bf4\u660e</b><br/>");
  out.println("<p>\u8fd9\u662f\u4e00\u4efd\u975e\u5e38\u95f4\u5355"
    + "\u7684\u8bf4\u660e\u4e66\u2026</p>");
  out.println("</body>");
  out.println("</html>");
]]></jsp:scriptlet>
</jsp:root>
```

When I opened HelpGB2312Unicode.jsp with IE, I saw Chinese characters correctly displayed on the screen. Remember I have Unicode Chinese fonts installed on my system. So option 2 works! But note that:

- Option 2 looks much simpler than option 1. No need to output HTML documents in binary mode.
- response.setContentType() must be called before any output statements.
- "out" is ready to use with the specified encoding schema embedded.
- The Chinese characters must be entered as Unicode codes, not GB2312 codes.

If you Chinese text is in GB2312 encoding format, you need to convert the text to Unicode codes in "\u" format. One good tool for this is native2ascii from JDK. Here is a sample command to convert HelpGB2312.html:

```
 \jdk\bin\native2ascii -encoding gb2312 HelpGB2312.html test.html
```

You could also enter non ASCII characters as Unicode codes in UTF-8 format. This is very easy to do, if you have a special text editor that supports Unicode UTF-8 encoding and input interface for your local language characters.

Entering Non ASCII Characters as Static HTML Text

Entering non ASCII characters as static HTML text is much harder than what I initially thought. There are many factors that should be considered:

- JSP page syntax - Using standard syntax or XML syntax.
- Encoding schema of the JSP page source code.
- Encoding schema of the converted Java source code.

- Encoding schema of the HTTP response.

In order to test out how to control those factors, I picked two simplified Chinese characters, and entered them in 7 different formats as a simple HTML paragraph:

```
<p>
GB2312-binary: ??=(0xCBB5C3F7)<br/>
GB2312-#xHEX: &#xCBB5;&#xC3F7;<br/>
GB2312-\uHEX: \uCBB5\uC3F7<br/>
Unicode-binary: ??=(0x8bf4660e)<br/>
Unicode-#xHEX: &#x8bf4;&#x660e;<br/>
Unicode-\uHEX: \u8bf4\u660e<br/>
Unicode-UTF8: ???=(0xE8AFB4E6988E)<br/>
</p>
```

Hex numbers are provided next to the binary codes, just in case if you have trouble to copy this file to your local system.

In the next 3 sections, I will put this paragraph into a regular HTML file, a JSP page with standard syntax, and a JSP page with XML syntax to see how Tomcat server will convert them into Java class files and in what incodings.

Static HTML Text - HTML Page

In the first test, the static text is inserted into a regular HTML file:

```
<html>
<!-- StaticGB2312.html
   Copyright (c) 2002 by Dr. Herong Yang
-->
<body>
<p>
GB2312-binary: ??=(0xCBB5C3F7)<br/>
GB2312-#xHEX: &#xCBB5;&#xC3F7;<br/>
GB2312-\uHEX: \uCBB5\uC3F7<br/>
Unicode-binary: ??=(0x8bf4660e)<br/>
Unicode-#xHEX: &#x8bf4;&#x660e;<br/>
Unicode-\uHEX: \u8bf4\u660e<br/>
Unicode-UTF8: ???=(0xE8AFB4E6988E)<br/>
</p>
</body>
</html>
```

Now view StaticGB2312.html with IE, and try to change the encoding schema in the View

menu. Results match my expectations except one area:

- Westen European (Windows): Unicode-#xHEX line shows up correctly. I wasn't expecting this, and had no idea why.
- Chinese Simplified (GB2312): GB2312-binary line shows up correctly.
- Unicode (UTF-8): Unicode-UTF8 line shows up correctly.

Since this is not a JSP, Tomcat will not convert it into a Java class file. I am using this test to validate that the codes are entered correctly.

Static HTML Text - JSP Page in Standard Syntax

In the second test, the static text is inserted into a JSP page in standard syntax:

```
<%@ page contentType="text/html; charset=gb2312" %>
<!-- StaticGB2312.jsp
    Copyright (c) 2002 by Dr. Herong Yang
-->
</html>
</body>
<p>
GB2312-binary: ??=(0xCBB5C3F7)<br/>
GB2312-#xHEX: &#xCBB5;&#xC3F7;<br/>
GB2312-\uHEX: \uCBB5\uC3F7<br/>
Unicode-binary: ??=(0x8bf4660e)<br/>
Unicode-#xHEX: &#x8bf4;&#x660e;<br/>
Unicode-\uHEX: \u8bf4\u660e<br/>
Unicode-UTF8: ???=(0xE8AFB4E6988E)<br/>
</p>
</body>
</html>
```

If you view this page in IE, you will that see both GB2312-binary line and Unicode-#xHEX line are displayed correctly. Here is the explanation:

- The "charset" value gb2312 in the page directive statement tells Tomcat to read this JSP files as GB2312 encoding. So GB2312-binary line is decoded correctly into Unicode codes.
- All other binary lines are decoded incorrectly, because they are not GB2312 codes.
- Uicode-#xHEX line is not decoded, because they are normal ASCII characters.
- When generating the Java class file, all strings are encoded as UTF-8. This is the default setting of Tomcat. You can change this in the conf/web.xml file.
- The "charset" value gb2312 also tells Tomcat to change the encoding to GB2312 on the "out" object, and the Conten-Type HTTP header, so the generated HTML document will

in GB2312 encoding.

To appove the above explanation, here is the related lines of the generated Java class file:

```
...
response.setContentType("text/html; charset=gb2312");
...
out.write("<p>\r\nGB2312-binary: ???=(0xCBB5C3F7)");
out.write("<br/>\r\nGB2312-#xHEX: &#xCBB5;&#xC3F7;");
out.write("<br/>\r\nGB2312-\\uHEX: \\uCBB5\\uC3F7");
out.write("<br/>\r\nUnicode-binary: ????=(0x8bf4660e)");
out.write("<br/>\r\nUnicode-binary: ----=(0x8bf4660e)");
out.write("<br/>\r\nUnicode-#xHEX: &#x8bf4;&#x660e;");
out.write("<br/>\r\nUnicode-\\uHEX: \\u8bf4\\u660e");
out.write("<br/>\r\nUnicode-UTF8: ??????=(0xE8AFB4E6988E)");
...
```

If you change the "charset" to utf-8, I am sure Unicode-UTF8 line will be displayed correctly. You know why.

By the way, "charset" can also be specified as "pageEncoding" in the "page" directive statement.

Static HTML Text - JSP Page in XML Syntax

In the third test, the static text is inserted into a JSP page in XML syntax:

```
<?xml version="1.0" encoding="gb2312"?>
<jsp:root xmlns:jsp="http://java.sun.com/JSP/Page"
  xmlns:c="http://java.sun.com/jstl/core" version="1.2">
<jsp:directive.page contentType="text/html; charset=gb2312"/>
<!-- StaticGB2312.jsp
    Copyright (c) 2002 by Dr. Herong Yang
-->
<html>
<body>
<p>
GB2312-binary: ??=(0xCBB5C3F7)<br/>
GB2312-#xHEX: &#xCBB5;&#xC3F7;<br/>
GB2312-\uHEX: \uCBB5\uC3F7<br/>
Unicode-binary: ----=(0x8bf4660e)<br/>
Unicode-#xHEX: &#x8bf4;&#x660e;<br/>
Unicode-\uHEX: \u8bf4\u660e<br/>
```

```
Unicode-UTF8: ???=(0xE8AFB4E6988E)<br/>
</p>
</body>
</html>
</jsp:root>
```

If you view this page with IE, you should will see that only Unicode-#xHEX line is displayed correctly. This is a big supprise to me:

- The XML parser in Tomcat is not deconding my JSP page with gb2312.
- My JSP page seems to be decoded with ISO-8859-1, Windows default encoding scheme.
- The 0x0e code in Unicode-binary line is causing trouble to Tomcat server, so I have to remove those binary codes.
- The Java class file is generated in UTF-8 encoding.
- The "out" object and the Content-Type header are set correctly to GB2312.
- The XML entity codes, #xHEX lines, are decoded into binary values. This is different than the standard syntax.

Here are the related lines of the generated Java class file:

```
...
response.setContentType("text/html; charset=gb2312");
...
out.write("<p>");
out.write("\nGB2312-binary: ????=(0xCBB5C3F7)");
out.write("<br/>");
out.write("\nGB2312-#xHEX: ");
out.write("??");
out.write("??");
out.write("<br/>");
out.write("\nGB2312-\\uHEX: \\uCBB5\\uC3F7");
out.write("<br/>");
out.write("\nUnicode-binary: ----=(0x8bf4660e)");
out.write("<br/>");
out.write("\nUnicode-#xHEX: ");
out.write("??");
out.write("??");
out.write("<br/>");
out.write("\nUnicode-\\uHEX: \\u8bf4\\u660e");
out.write("<br/>");
out.write("\nUnicode-UTF8: ??????=(0xE8AFB4E6988E)");
out.write("<br/>");
out.write("</p>");
....
```

I have tried to change charset to UTF-8, but it did not work. JSP pages in XML syntax are always decoded as ISO-8859-1. May be there is a setting somewher to control this, but I don't know.

Supporting Characters from Multiple Languages

If you planning to write a page that has characters from multiple language encodings. you have to use Unicode codes and UTF-8 HTML document encoding. Here is an example with characters from two encodings: GB2312 and Big5.

```
<?xml version="1.0"?>
<jsp:root xmlns:jsp="http://java.sun.com/JSP/Page"
  xmlns:c="http://java.sun.com/jstl/core" version="1.2">
<!-- HelpUnicodeUTF8.jsp
    Copyright (c) 2004 by Dr. Herong Yang
-->
<jsp:scriptlet><![CDATA[
  response.setContentType("text/html; charset=utf-8");
  out.println("<meta http-equiv=\"Content-Type\""
    + " content=\"text/html; charset=utf-8\"/>");
  out.println("<body>");
  out.println("<b>\u8bf4\u660e</b><br/>");
  out.println("<p>\u8fd9\u662f\u4e00\u4efd\u975e\u5e38\u95f4\u5355"
    + "\u7684\u8bf4\u660e\u4e66\u2026</p>");
  out.println("<b>\u8aaa\u660e</b><br/>");
  out.println("<p>\u9019\u662f\u4e00\u4efd\u975e\u5e38\u9593\u55ae"
    + "\u7684\u8aaa\u660e\u66f8\u2026</p>");
  out.println("</body>");
  out.println("</html>");
]]></jsp:scriptlet>
</jsp:root>
```

View this page with IE, you should see the same message appear twice, one as simplified Chinese, and the other as tranditional Chinese.

Conclusion

As you can see from my notes in the previous sections, localizing or internationalizing JSP pages is not an easy task. My recommendations are:

- Avoid using static text. Put the entire page under a scriptlet, so all text messages are generated from Java statements.
- Using Unicode codes in UTF-8 format or \uHEX format for string literals. It allows to

support characters in all local languages in a single encoding.

- Use UTF-8 as the HTML document encoding instead of encodings of a particular local language, like GB2312. This may cause problems for users on locale systems where Unicode fonts are not supported. But more and more locale systems are supporting Unicode and UTF-8 encoding.
- I still don't know how to control the source code encoding of JSP pages in XML syntax.

JSP Performance

Calculating Prime Numbers

The first area I want to test for performance is integer arithmetic calculations. The following JSP page calculates prime number starting from number 3, and repeats the test many times.

```
<?xml version="1.0"?>
<jsp:root xmlns:jsp="http://java.sun.com/JSP/Page" version="1.2">
<!-- PrimeNumbers.jsp
    Copyright (c) 2003 by Dr. Herong Yang
-->
<jsp:directive.page contentType="text/html"/>
<jsp:directive.page import="java.util.*"/>
<jsp:scriptlet><![CDATA[
  int[] primes = new int[1000];
  int numberOfTests = 100;
  int numberOfPrimes = 1000;
  long t1 = System.currentTimeMillis();
  for (int nTest=1; nTest<=numberOfTests; nTest++) {
    // Getting prime numbers
    int nPrime = 0;
    int i = 2;
    while (nPrime < numberOfPrimes) {
      i = i + 1;
      int j = 2;
      boolean isPrime = true;
      while (j<i && isPrime) {
        isPrime = i % j > 0;
        j = j + 1;
      }
      if (isPrime) {
        nPrime = nPrime + 1;
        primes[nPrime-1] = i;
      }
    }
  }
  long t2 = System.currentTimeMillis();
```

```
  long t = t2 - t1;

  // Displaying the results
  out.println("<html><body>");
  out.println("<b>Performace Information:</b><br/>");
  out.println("Number of tests = " + numberOfTests + "<br/>");
  out.println("Time = " + (t/1000) + " seconds.<br/>");
  out.println("<b>" + numberOfPrimes + " prime numbers:</b><br/>");
  for (int n = 1; n <= numberOfPrimes; n++) {
    out.println(primes[n-1] + ", ");
  }
  out.println("</body></html>");
]]></jsp:scriptlet>
</jsp:root>
```

I run this page, and got the following result. It tells me the page is working correctly.

```
Performace Information
Number of tests = 1
Time = 0 seconds
First 1000 Prime numbers:
3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67, ...
```

By changing the controlling parameters, I was able to get some results:

Cases	Number of Tests	Number of Primes	Debug Mode	Time (sec)	Notes
1.	1	1000	Yes	25	ASP with IIS 5.0
2.	1	1000	No	25	ASP with IIS 5.0
3.	1	1000	?	0	JSP with Tomcat 4.1.18
4.	10	1000	?	2	JSP with Tomcat 4.1.18
5.	100	1000	?	22	JSP with Tomcat 4.1.18
6.	100	1000	?	21	JVM HotSpot 1.3.1

So this tells us that JSP pages are 100 times faster than ASP pages for integer calculations.

Response Time of "Hello" Page

The next area I want test is total response time of ASP pages. To do this, I wrote the following Java program. This program is doing a single HTTP request to a specifield Web page, and repeating this for many times.

```
/**
 * HttpResponseTest.java
 * Copyright (c) 2002 by Dr. Herong Yang. All rights reserved.
 */
import java.io.*;
import java.net.*;
public class HttpResponseTest {
  public static void main(String[] args) {
    int numberOfTests = 1;
    if (args.length > 0) numberOfTests
      = Integer.valueOf(args[0]).intValue();
    long t1 = System.currentTimeMillis();
    String result = "";
    for (int nTest=1; nTest<=numberOfTests; nTest++) {
      result = test(args);
    }
    long t2 = System.currentTimeMillis();
    long t = t2 - t1;
    PrintStream out = System.out;
    out.println("Performace Information:");
    out.println("   Number of tests = " + numberOfTests);
    out.println("   Time = " + (t/1000) + " seconds.");
    out.println("Rerulst of Last Test:");
    out.println(result);
  }
  public static String test(String[] args) {
    String path = "/index.html";
    int port = 80;
    String host = "localhost";
    if (args.length > 1) path = args[1];
    if (args.length > 2) port
      = Integer.valueOf(args[2]).intValue();
    if (args.length > 3) host = args[3];
    String result = "";
    try {
      Socket c = new Socket(host,port);
      BufferedWriter w = new BufferedWriter(new OutputStreamWriter(
        c.getOutputStream()));
      BufferedReader r = new BufferedReader(new InputStreamReader(
        c.getInputStream()));
      String m = "GET "+path;
      w.write(m,0,m.length());
      w.newLine();
      w.flush();
```

```
        while ((m=r.readLine())!= null) {
          result = result + m + "\n";
        }
        w.close();
        r.close();
        c.close();
      } catch (IOException e) {
        System.err.println(e.toString());
      }
      return result;
    }
  }
```

Now compile this program and try it with the following "Hello" JSP page, hello.jsp:

```
<html><body>
<% out.println("Hello world!"); %>
</body></html>
```

you will get something similar to this:

```
\local\j2sdk1.4.1_01\bin\java -cp . HttpResponseTest 1 /hello.jsp 8080
Performace Information:
  Number of tests = 1
  Time = 1 seconds.
Rerulst of Last Test:
<html><body>
Hello world!

</body></html>
```

I repeated the tests by changes the controlling parameters. The following table shows the results comparing with similar tests I did with other technologies:

Cases	Number of Tests	Debug Mode	Time (Sec)	Note
1.	1000	No	2	Static text with IIS 5.0
2.	2000	No	4	Static text with IIS 5.0
3.	1000	No	6	ASP page with IIS 5.0
4.	2000	No	11	ASP page with IIS 5.0
5.	1000	?	7	Static text with Tomcat 4.1.18
6.	2000	?	15	Static text with Tomcat 4.1.18
7.	1000	?	8	JSP page with Tomcat 4.1.18

8. 2000 ? 16 JSP page with Tomcat 4.1.18

Conclusion:

- So the performance cost of writing text through ASP statements is 2 times slower than the static pages.
- Tomcat is 3 times slower than IIS when serving static pages.
- Tomcat is also slower than IIS when serving dynamic text.

JSP Standard Tag Libraries (JSTL)

What is JSTL?

JSTL (JSP Standard Tag Libraries) is a collection of JSP custom tags developed by Java Community Process, www.jcp.org. The reference implementation is developed by the Jakarta project, jakarta.apache.org.

The latest version of JSTL is JSTL 1.1, which requires a JSP container that supports the Java Servlet 2.4 and JavaServer Pages 2.0 specifications. Jakarta Tomcat 5 supports these specifications.

The previous version is JSTL 1.0, which requires a JSP container that supports the Java Servlet 2.3 and JavaServer Pages 1.2 specifications. Jakarta Tomcat 4 supports these specifications.

Since I have Tomcat 4.1.18 installed on my machine, I will talk about JSTL 1.0 only in this section.

The goal of JSTL, as described in the specification, is to help simplify JavaServer Pages page authors' lives. To achieve this goal, JSTL has provided custom tags for many common JSP page authoring tasks that require scripting statements to manipulate server side dynamic data.

JSTL offers tags through 4 libraries:

- core - Basic scripting functions
- xml - XML processing
- fmt - Internationalization of formatting
- sql - Data base accessing

Installing JSTL 1.0 Implementation - Standard Taglib 1.0.4

Standard Taglib 1.0.4 is Jakara Taglibs's open-source implementation of the JSP Standard Tag Library (JSTL) 1.0. I did the following to download the last release of Standard Taglib 1.0.4:

- Go to http://apache.towardex.com/jakarta/taglibs/standard-1.0/ and download jakarta-taglibs-standard-current.zip.
- Unzip it to \local directory, and read \local\jakarta-taglibs-standard-1.0.4\README.
- Then create a new directory in Tomcat server: "mkdir \local\jakarta-tomcat-4.1.18\webapps\ROOT\WEB-INF\lib" and copy jar files to there:

"copy \local\jakarta-taglibs-standard-1.0.4\lib*.jar
\local\jakarta-tomcat-4.1.18\webapps\ROOT\WEB-INF\lib".

- Restart Tomcat, and you are ready to try Taglib 1.0.4.

"Hello world!" with JSTL

To understand how JSTL works, let's try a very simple example, using JSTL to display "Hello world!". Here is my JSP source code, hello_jstl.jsp:

```
<%@ taglib uri="http://java.sun.com/jstl/core" prefix="c" %>
<html><body>
<c:out value="Hello world!"/>
</body></html>
```

Save it to \local\jakarta-tomcat-4.1.18\webapps\ROOT, and run IE with url: http://localhost:8080/hello_jstl.jsp. Guess what? You will receive crashing page with an error message like: "javax.servlet.ServletException: Cannot inherit from final class".

So what happened? I really don't know. My guess is that some of the jar files from Taglib 1.0.4 are not compatible with Tomcat 4.1.18. To approve this, I removed all Taglib jar files, except standard.jar and jstl.jar, from \local\jakarta-tomcat-4.1.18\webapps\ROOT\WEB-INF\lib. I restarted Tomcat and ran IE again with url: http://localhost:8080/hello_jstl.jsp. I got the prefect message "Hello world!" in the IE window!

Now let's see the Servlet class generated by Tomcat server based on my JSP page with the "c:out" tag. The Servlet class is located at \local\jakarta-tomcat-4.1.18\work\standalone\localhost_\hello_jstl_jsp.java:

```
package org.apache.jsp;

import javax.servlet.*;
import javax.servlet.http.*;
import javax.servlet.jsp.*;
import org.apache.jasper.runtime.*;

public class hello_jstl_jsp extends HttpJspBase {


  private static java.util.Vector _jspx_includes;

  private org.apache.jasper.runtime.TagHandlerPool
    _jspx_tagPool_c_out_value;

  public hello_jstl_jsp() {
```

```
  _jspx_tagPool_c_out_value =
    new org.apache.jasper.runtime.TagHandlerPool();
}

public java.util.List getIncludes() {
  return _jspx_includes;
}

public void _jspDestroy() {
  _jspx_tagPool_c_out_value.release();
}

public void _jspService(HttpServletRequest request,
   HttpServletResponse response)
     throws java.io.IOException, ServletException {

  JspFactory _jspxFactory = null;
  javax.servlet.jsp.PageContext pageContext = null;
  HttpSession session = null;
  ServletContext application = null;
  ServletConfig config = null;
  JspWriter out = null;
  Object page = this;
  JspWriter _jspx_out = null;


  try {
    _jspxFactory = JspFactory.getDefaultFactory();
    response.setContentType("text/html;charset=ISO-8859-1");
    pageContext = _jspxFactory.getPageContext(this, request, response,
                null, true, 8192, true);
    application = pageContext.getServletContext();
    config = pageContext.getServletConfig();
    session = pageContext.getSession();
    out = pageContext.getOut();
    _jspx_out = out;

    out.write("\r\n");
    out.write("<!--%@ taglib uri=\"http://java.sun.com/jstl/core\""
      +" prefix=\"c\" %-->\r\n");
    out.write("<html>");
    out.write("<body>\r\n");
    if (_jspx_meth_c_out_0(pageContext))
      return;
```

```
    out.write("\r\n");
    out.write("</body>");
    out.write("</html>");
  } catch (Throwable t) {
    out = _jspx_out;
    if (out != null && out.getBufferSize() != 0)
      out.clearBuffer();
    if (pageContext != null) pageContext.handlePageException(t);
  } finally {
    if (_jspxFactory != null)
      _jspxFactory.releasePageContext(pageContext);
  }
}

 private boolean _jspx_meth_c_out_0(javax.servlet.jsp.PageContext
   pageContext)
       throws Throwable {
  JspWriter out = pageContext.getOut();
  /* ----  c:out ---- */
  org.apache.taglibs.standard.tag.el.core.OutTag _jspx_th_c_out_0
    = (org.apache.taglibs.standard.tag.el.core.OutTag)
      _jspx_tagPool_c_out_value.get(
      org.apache.taglibs.standard.tag.el.core.OutTag.class);
  _jspx_th_c_out_0.setPageContext(pageContext);
  _jspx_th_c_out_0.setParent(null);
  _jspx_th_c_out_0.setValue("Hello world!");
  int _jspx_eval_c_out_0 = _jspx_th_c_out_0.doStartTag();
  if (_jspx_th_c_out_0.doEndTag() ==
    javax.servlet.jsp.tagext.Tag.SKIP_PAGE)
    return true;
  _jspx_tagPool_c_out_value.reuse(_jspx_th_c_out_0);
  return false;
 }
}
```

As you can see, the "c:out" tag is replaced by method call. The method then interacts with the class org.apache.taglibs.standard.tag.el.core.OutTag, where the "out" tag is implemented. The implementation is probably very simple, may be just an out.write() statement.

JSTL in XML Style JSP Pages

Since I like to write JSP in XML style, I have to find out how to use JSTL in XML style JSP pages. It took me some time to figure this. Here is an example code, hello_jstl_xml.jsp:

```
<?xml version="1.0"?>
<jsp:root xmlns:jsp="http://java.sun.com/JSP/Page"
  xmlns:c="http://java.sun.com/jstl/core" version="1.2">
<!-- hello_jstl_xml.jsp
    Copyright (c) 2003 by Dr. Herong Yang
-->
<jsp:directive.page contentType="text/html"/>
<html><body>
<jsp:scriptlet>out.println("Hello world!");</jsp:scriptlet>
<br/>
<c:out value="Hello world! - from c:out"/>
</body></html>
</jsp:root>
```

The trick is to convert the tablib to a name space attribute in the jsp:root element.

JSTL Requirements

The requirements to use JSTL are:

- A JSP server - Tomcat from apache.org.
- A JSTL implementation - Tablib from apache.org, jstl.jar and standard.jar.
- A JSTL document - JSTL specification from jcp.org.
- JSP pages with JSTL tags - You write them.

JSTL - Syntax and Expression Language

This chapter explains:

- Basics on JSTL syntax.
- Basics on Expression Language.
- Literal data and named variables.
- Basic operators and operations.
- Accessing collection elements and object properties.
- ExpExample.jsp - Expression examples JSP page.
- pageContext attributes and JSTL top level identifiers.

JSTL Syntax

JSTL syntax is very simple. JSTL allows you to write tags in JSP pages. Each tag is actually called an action. Every action must be written as an XML element. The syntax of an JSTL action XML element is something like these:

```
<p:tag attribute="text_only"/>
<p:tag>
  xml_body
</p:tag>
<p:tag attribute="text_only">
  xml_body
</p:tag>
<p:tag attribute="${expression}">
  xml_body
</p:tag>
<p:tag attribute="text${expression}text${express}..." ...>
  xml_body
</p:tag>
```

As you can see, there are a number variations in the syntax:

- An action can be an empty or non-empty XML element.
- An action can have zero, one, or many attributes.
- Attribute values can be text only, or mixed with expressions.
- An express is always written in the format of ${expression}.

The expressions must be written by folowing rules defined by the expression language.

Examples of JSTL actions:

```
<c:out value="Hello world!"/>
<c:if test="${1+1==2}">
  Always true.
</c:if>
<c:set var="message" value="Hello world!"/>
<c:out value="${message}"/>
```

Expression Language

Since expression will be used in most of the JSTL actions, let's look at the express language before individual JSTL actions.

JSTL expression language is inspired by both ECMAScript and XPath expression language. It is simple, and supporting the following features:

- Literal data and named variables.
- Logical, relational and arithmetic operations.
- A set of implicit objects.
- Nested properties and accessors to collections.

Examples of JSTL expressions:

```
<c:out value="${1+1==2}"/>
<c:out value="${1+1}"/>
<c:out value="${1/3}"/>
<c:out value="${1.0/3.0}"/>
<c:out value="${message}"/>
<c:out value="${pageContext.request.class.name}"/>
<c:out value="${pageContext.request.method}"/>
<c:out value="${pageContext.request.requestedSessionId}"/>
<c:out value="${pageContext.request.cookies[0].name}"/>
<c:out value="${quantity*price < 100.0 && country=='USA'}"/>
```

Literal Data and Named Variables

As in all computer language, expression starts with literal data and variables. JSTL expression language supports 5 types of literal data:

- Boolean - true and false. Same as Java boolean.

- Integer - like 9999, -3, and 0. Same as Java long.
- Floating point number - like, 1.0, 3.14159, and -1.0e-3. Same as Java double.
- String - like, 'USA', "USA", or 'Herong\'s notes. Close to Java String.
- Null - null. Same as Java null.

The literal data rules are easy to understand, with a couple of exceptions:

- No literal data for character data type.
- String can be quoted by single quote (').

Named variables are coming from two sources:

- Variables defined by the JSTL "set" action.
- Variables provided as attributes in the pageContext object prepared by other server side codes, like useBean and scriptlet JSP elements.

Variables provided as pageContext attributes will have their original Java types, like int, float, char, or Object. But all JSTL operations will be carried out in one of the 5 types of literal data. Operators of other types will be converted before the operation.

Type conversion rules:

- To string - Using the Java rules, except that null will be converted to "".
- To boolean - Using the Java rules, except that null or "" will be converted to false.
- To integer - Using the Java rules, except that null or "" will be converted to 0.
- To floating porint number - Using the Java rules, except that null or "" will be converted to 0.0.
- To object - Using the Java rules, except that "" will be converted to null.

Basic Operators and Operations

JSTL supports all logical, relational and arithmetic operators supported in Java, with exceptions:

- "eq", "ne", "lt", "gt", "le", and "ge" could be used as relational operators.
- "and", "or" and "not" could be used as logical operators.

Implicit Objects

JSTL has a set of predefined objects accessible by the following variable names:

- pageContext - the PageContext object.
- pageScope - a Map that maps page-scoped attribute names to their values.
- requestScope - a Map that maps request-scoped attribute names to their values.
- sessionScope - a Map that maps session-scoped attribute names to their values.

- applicationScope - a Map that maps application-scoped attribute names to their values.
- param - a Map that maps parameter names to a single String parameter value (obtained by calling ServletRequest.getParameter(String)).
- paramValues - a Map that maps parameter names to a String[ ] of all values for that parameter (obtained by calling ServletRequest.getParameterValues(String)).
- header - a Map that maps header names to a single String header value (obtained by calling ServletRequest.getheader(String)).
- headerValues - a Map that maps header names to a String[ ] of all values for that parameter (obtained by calling ServletRequest.getHeaders(String)).
- cookie - a Map that maps cookie names to a single Cookie (obtained by calling HttpServletRequest.getCookie(String)).
- initParam - a Map that maps a parameter names to a single String parameter value (obtained by calling ServletRequest.getInitParameter(String)).

Accessing Collection Elements and Object Properties

JSTL supports reference operations to collection elements and object properties by using "." and "[]" operators. But they are used in the way as ECMAScript, which is very different than Java. Here is the main steps of the evaluation process:

- 0. Check syntax to only allow: "identifier_a.identifier_b" and "identifier_a[expression_b]".
- 1. Replacing "." operator by "[]" operator. So convert "identifier_a.identifier_b" to "identifier_a.['identifier_b']".
- 2. Evaluate the second operand. So convert "identifier_a[expresion_b]" to "identifier_a[value_b]".
- 3. If the first operand matches no existing object name, return null.
- 4. If the second operand evaluates to null, return null.
- 5. If the first operand reprents an array, try to finish the operation as "[]". So try to evaluate "identifier_a[value_b]" as is.
- 6. If the first operand reprents a map, try to finish the operation with a call to get(). So try to evaluate "identifier_a[value_b]" as "identifier_a.get(value_b)".
- 7. If the first operand represents an object of orther types, try to finish the operation as a Java bean property accessing operation. So convert "identifier_a[value_b]" to "identifier_a[identifier_b]", then evaluate it as "identifier_a.get'Identifier_b'()".
- 8. If step 7 failed, try to finish the operation as an object member variable. So evaluate "identifier_a[identifier_b]" as "identifier_a.identifier_b". I am note so sure about this. Needs further research.

As you can see, this process is very complex. But it does make page author's life easier by putting a lot of intelligence behind this operation. But it also brings a lot of confusion when you debug the code.

The JSTL specification about the "." and "[]" operators is quoted below as reference. But I believe my description is much easier to understand.

The expression language follows ECMAScript in unifying the treatment of the "." and "[]" operators.

expr-a.identifier-b is equivalent to a["identifier-b"]; that is, the identifier identifier-b is used to construct a literal whose value is the identifier, and then the "[]" operartor is used with that value.

To evaluate expr-a[expr-b]:
. Evaluate expr-a into value-a
. if value-a is null, return null.
. Evaluate expr-b into value-b
. if value-b is null, return null.
. if value-a is a Map, List or array
  . if value-a is a Map
    . if !value-a.containsKey(value-b) then return null.
    . otherwise, return value-a.get(value-b)
  . if value-a is a List or array
    . coerce value-b to int (using coercing rules);
      . if coercion couldn't be performed: error
    . then, if value-a.get(value-b) or Array.get(value-a, value-b)
      throws ArrayIndexOutOfBoundsException or
      IndexOutOfBoundsException: return null
    . otherwise, if value-a.get(value-b) or Array.get(value-a, value-b)
      throws other exception, error
    . otherwise, return value-a.get(value-b) or Array.get(value-a,
      value-b), as appropriate.
. Otherwise (a JavaBeans object), coerce value-b to String
  . If value-b is a readable property of value-a
    . if getter throws an exception: error
    . otherwise; return result of getter call
  . otherwise: error

ExpExample.jsp - Expression Example Page

To check my understanding of the expression language, I wrote the following JSP to show some expression examples.

```
<?xml version="1.0"?>
<jsp:root xmlns:jsp="http://java.sun.com/JSP/Page"
  xmlns:c="http://java.sun.com/jstl/core" version="1.2">
<!-- ExpExample.jsp
    Copyright (c) 2003 by Dr. Herong Yang
-->
```

```
<jsp:directive.page contentType="text/html"/>
<html><body>
<p>JSTL Expression Examples:</p>
<c:out value="1. ${1+1==2}"/><br/>
<c:out value="2. ${1+1}"/><br/>
<c:out value="3. ${1/3}"/><br/>
<c:out value="4. ${1.0/3.0}"/><br/>
<c:out value="5. ${quantity*price lt 100.0 and country=='USA'}"/><br/>
<c:set var="message" value="Hi there!"/>
<c:out value="6. ${message}"/><br/>
<c:out value="7. ${'message'}"/><br/>
<c:out value="10. ${pageContext.request.method}"/><br/>
<c:out value="11. ${pageContext.request[method]}"/><br/>
<c:out value="12. ${pageContext.request['method']}"/><br/>
<!-- c:out value="13. ${pageContext.request.'method'}"/><br/ -->
<!-- c:out value="14. ${pageContext.'request'.method}"/><br/ -->
<c:out value="15. ${pageContext['request']['method']}"/><br/>
<c:out value="16. ${pageContext['request'].method}"/><br/>
<!-- c:out value="20. ${pageContext.request.cookies.length}"/><br/ -->
<c:out value="21. ${pageContext.request.cookies[0].name}"/><br/>
<c:out value="22. ${pageContext.request.cookies[0].value}"/><br/>
<c:out value="23. ${pageContext.request.cookies[0]}"/><br/>
<!-- c:out value="24. ${pageContext.request.cookies.0}"/><br/ -->
<!-- c:out value="25. ${pageContext.request.cookies.0.name}"/><br/ -->
<c:out value="26. ${pageContext.request.cookies['0']}"/><br/>
<jsp:scriptlet>double pi = 3.14159;</jsp:scriptlet>
<c:out value="30. ${pi}"/><br/>
<jsp:scriptlet>
  double[] list = new double[3];
  list[0] = 9.99;
</jsp:scriptlet>
<c:out value="31. ${list[0]}"/><br/>
<c:out value="32. ${pageContext.request.class.name}"/><br/>
<c:out value="33. ${pageContext[request[class[name]]]}"/><br/>
<c:out value="34. ${out.class.name}"/><br/>
<c:out value="35. ${pageContext.out.class.name}"/><br/>
<!-- c:out value="36. ${sessionScope.now}"/><br/ -->
<jsp:scriptlet>
  session.setAttribute("now", new java.util.Date());
</jsp:scriptlet>
<c:out value="37. ${sessionScope.now.time}"/><br/>
</body></html>
</jsp:root>
```

Here is the output this page. But you read the output, I want you take guess on the resulting values my expression examples would give and compare them with the output.

JSTL Expression Examples:

1. true
2. 2
3. 0.3333333333333333
4. 0.3333333333333333
5. false
6. Hi there!
7. message
10. GET
11.
12. GET
15. GET
16. GET
21. JSESSIONID
22. FAC70AB7F4107E5EC81A14B1C080FC79
23. javax.servlet.http.Cookie@352d87
26. javax.servlet.http.Cookie@352d87
30.
31.
32. org.apache.coyote.tomcat4.CoyoteRequestFacade
33.
34.
35. org.apache.jasper.runtime.JspWriterImpl
37. 1065795381165

Do you have any supprises when you compare your guess with output? I have some explanations on the output to help you:

- Example 3 shows you that the result of integer operations is not converted by to integer.
- Example 5 confirms that null is converted to false.
- Example 7 shows ycu that value will be converted back to an identifier (variable name).
- Examples 10 and 12 show you two correct ways of accessing object properties.
- Examples 11 is understandable, because an expression is expected in side "[]" and "method" without quotes is an identifier and it matches no existing objects (variable names).
- Examples 13 and 14 have syntax errors, because the "." operator only takes identifiers (names) as operands.
- Exampels 15 and 16 are different ways of writting nested "." operations.
- Exampels 24 and 25 have syntax errors, because "0" is not allowed an identifier.
- Exampels 30 and 31 tell us that variables declared by scriptlet are not available to JSTL.

- Exampel 32 is working because Java offers Object.getClass().getName().
- Exampel 33 is not working because "[name]" evaluates to null.
- Exampels 34 and 35 show you that the out is not directly accessible, but it is an accessible property of pageContext.
- Exampels 36 and 37 show you that new elements can be added to implicit map object, sessionScope.

Exercise: Write an example JSP page to show how to create a JavaBean object, and how to access the properties of this object with JSTL expressions.

pageContext Attributes and JSTL Top Level Identifiers

Based the rules and the examples in the previous sections, we can easily conclude that:

- JSTL top level identifiers (variables) are pageContext attributes.
- pageContext attributes are JSTL top level identifiers (variables).

Here is a sample code to show you that you can mix variables and pageContext attributes any way you want:

```
<?xml version="1.0"?>
<jsp:root xmlns:jsp="http://java.sun.com/JSP/Page"
  xmlns:c="http://java.sun.com/jstl/core" version="1.2">
<!-- ExpVariable.jsp
    Copyright (c) 2003 by Dr. Herong Yang
-->
<jsp:directive.page contentType="text/html"/>
<html><body>
<c:set var="message" value="Hi there!"/>
<c:out value="1. ${message}"/><br/>
2. <jsp:expression>pageContext.findAttribute("message")
  </jsp:expression><br/>

<jsp:scriptlet><![CDATA[
  String s = "Hello world!";
  pageContext.setAttribute("hello", s, PageContext.PAGE_SCOPE);
]]></jsp:scriptlet>
<c:out value="3. ${hello}"/><br/>

<jsp:useBean id="today" class="java.util.Date"/>
<c:out value="4. ${today}"/><br/>

<jsp:scriptlet><![CDATA[
  java.util.Date d = new java.util.Date();
```

```
   d.setTime(d.getTime()+24*60*60*1000);
   pageContext.setAttribute("tomorrow", d, PageContext.PAGE_SCOPE);
]]></jsp:scriptlet>
<c:out value="5. ${tomorrow}"/><br/>

<jsp:setProperty name="today" property="time" value="1000000000000"/>
<c:out value="6. ${today}"/><br/>
<c:out value="7. ${today.time}"/><br/>

</body></html>
</jsp:root>
```

Here is the output this page:

1. Hi there!
2. Hi there!
3. Hello world!
4. Sat Jul 12 14:06:16 EDT 2003
5. Sun Jul 13 14:06:16 EDT 2003
6. Sat Sep 08 21:46:40 EDT 2001
7. 1000000000000

Note that:

- useBean element also adds an attribute to pageContext. For more information see chapter "Using JavaBean Classes".

Exercise: Since there many methods to add an attribute to pageContext, useBean, c:set, and pageContext.setAttribute, write a simple program to show what happens if the same attribute name is used by different methods.

JSTL - Core Library

JSTL Core Library

JSTL core libary can be introduced to a JSP page with:

```
<jsp:root xmlns:jsp="http://java.sun.com/JSP/Page"
  xmlns:c="http://java.sun.com/jstl/core" version="1.2">
```

It provides the following basic scripting functions:

```
<c:out value="..."/>
<c:set var="..." value="..."/>
<c:if test="...">body</c:if>
<c:choose>body</c:choose>
<c:forEach items="...">body</c:forEach>
<c:forTokens items="..." delims="...">body</c:forTokens>
```

c:out Action

```
<c:out value="text_mixed_with_expressions"/>
```

The "value" attribute will be evaluated and the resulting value will be converted into a string which will be inserted into the HTTP response. The c:out action serves similar purposes as the JSP expression element.

c:set Action

```
<c:set var="name" value="expression"/>
```

The "var" attribute apecifies a variable name, which will be declared and assigned with the value resulted from the "value" attribute. The c:set action serves similar purposes as the Java assignement statement.

c:if Action

```
<c:if test="expression"/>
body
</c:if>
```

If the "test" attribute is evaluated to true, body will be processed. The c:if action serves similar purposes as the Java if statement.

c:choose Action

```
<c:choose>
 <c:when test="expression">
  body
 </c:when>
 <c:when test="expression">
  body
 </c:when>
 ...
 <c:otherwise>
  body
 </c:otherwise>
</c:choose>
```

The body of the first c:when action that has the "test" attribute evaluated to true will be processed. If all "test" attributes are evaluated to false, the body of the c:otherwise action will be processed. The c:choose action serves similar purposes as the Java if-else statment.

c:forEach Action

```
<c:forEach var="name" items="expression"
  begin="expression" end="expression" step="expression">
 body
</c:forEach>
```

If the "items" attribute is specified, it will be used to identify an array or collection object, the elements in the array or collection will be iterated. At each iteration, the current element will assign to a named variable specified in the "var" attribute, and body will be processed. If the "begin", "end" or "step" attribute is specified, it will be used to restrict the beginning element, the ending element, or the step size of the iteration.

If the "items" attribute is not specified, an index integer will be used to iterate from the "begin" value to the "end" value stepping with the "step" value.

The c:forEach action serves similar purposes as the Java for statement. But there is no break

mechanism.

c:forTokens Action

```
<c:forTokens var="name" items="expression" delims="expression"
  begin="expression" end="expression" step="expression">
 body
</c:forTokens>
```

The string specified by the "items" attribute will be tokenized with the delimiter specified by the "delims" attribute. Then resulting tokens will be iterated. At each iteration, the current token will assign to a named variable specified in the "var" attribute, and body will be processed. If the "begin", "end" or "step" attribute is specified, it will be used to restrict the beginning element, the ending element, or the step size of the iteration.

JSTL Core Example - JstlObjects.jsp

As my first JSTL core example, JstlObjects.jsp, is to use c:forEach to browse through all the implicit objects, and c:forTekons to break the class path into multiple items.

```
<?xml version="1.0"?>
<jsp:root xmlns:jsp="http://java.sun.com/JSP/Page"
  xmlns:c="http://java.sun.com/jstl/core" version="1.2">
<!-- JstlObjects.jsp
    Copyright (c) 2003 by Dr. Herong Yang
-->
<jsp:directive.page contentType="text/html"/>
<html><body>
<p>Browsing all the JSTL implicit objects:</p>
<p>"pageContext":</p>
<c:out value="${pageContext}"/><br/>

<p>"pageScope":</p>
<c:forEach items="${pageScope}" var="entry">
 <c:out value="${entry}"/><br/>
</c:forEach>

<p>"requestScope":</p>
<c:forEach items="${requestScope}" var="entry">
 <c:out value="${entry}"/><br/>
</c:forEach>

<p>"sessionScope":</p>
```

```
<c:forEach items="${sessionScope}" var="entry">
 <c:out value="${entry}"/><br/>
</c:forEach>

<p>"applicationScope":</p>
<c:forEach items="${applicationScope}" var="entry">
 <c:out value="${entry}"/><br/>
</c:forEach>

<p>"param":</p>
<c:forEach items="${param}" var="entry">
 <c:out value="${entry}"/><br/>
</c:forEach>

<p>"paramValues":</p>
<c:forEach items="${paramValues}" var="entry">
 <c:out value="${entry}"/><br/>
</c:forEach>

<p>"header":</p>
<c:forEach items="${header}" var="entry">
 <c:out value="${entry}"/><br/>
</c:forEach>

<p>"headerValues":</p>
<c:forEach items="${headerValues}" var="entry">
 <c:out value="${entry}"/><br/>
</c:forEach>

<p>"cookie":</p>
<c:forEach items="${cookie}" var="entry">
 <c:out value="${entry}"/><br/>
</c:forEach>

<p>"initParam":</p>
<c:forEach items="${initParam}" var="entry">
 <c:out value="${entry}"/><br/>
</c:forEach>

<p>Class path list:</p>
<c:forTokens
 items="${applicationScope['org.apache.catalina.jsp_classpath']}"
 delims=";" var="entry">
 <c:out value="${entry}"/><br/>
```

```
</c:forTokens>

</body></html>
</jsp:root>
```

Output of JstlObjects.jsp, slightly reformatted:

```
Browsing all the JSTL implicit objects:

"pageContext":

org.apache.jasper.runtime.PageContextImpl@4ed821

"pageScope":

javax.servlet.jsp.jspApplication=org.apache.catalina.core.ApplicationCon
javax.servlet.jsp.jspSession=org.apache.catalina.session.StandardSession
org.apache.taglibs.standard.ImplicitObjects=org.apache.taglibs.standard.
javax.servlet.jsp.jspOut=org.apache.jasper.runtime.JspWriterImpl@7c3828
javax.servlet.jsp.jspResponse=org.apache.coyote.tomcat4.CoyoteResponseFa
javax.servlet.jsp.jspRequest=org.apache.coyote.tomcat4.CoyoteRequestFaca
javax.servlet.jsp.jspConfig=org.apache.catalina.core.StandardWrapperFaca
javax.servlet.jsp.jspPage=org.apache.jsp.JstlObjects_jsp@27538
javax.servlet.jsp.jspPageContext=org.apache.jasper.runtime.PageContextIm

"requestScope":

"sessionScope":

"applicationScope":

javax.servlet.context.tempdir=D:\local\jakarta-tomcat-4.1.18\work\Standa
org.apache.catalina.WELCOME_FILES=[Ljava.lang.String;@628b8d
org.apache.catalina.resources=org.apache.naming.resources.ProxyDirContex
org.apache.catalina.jsp_classpath=/D:/local/jakarta-tomcat-4.1.18/webapp

"param":

"paramValues":

"header":

accept-language=en-us
connection=Keep-Alive
```

cookie=JSESSIONID=81C98734DD4F0E3F75608D6E3B1D83D0
accept=*/*
host=localhost:8080
user-agent=Mozilla/4.0 (compatible; MSIE 6.0; MSNIA; Windows NT 5.0; Q31

"headerValues":

accept-language=[Ljava.lang.String;@5dc721
connection=[Ljava.lang.String;@3c59aa
cookie=[Ljava.lang.String;@22da07
accept=[Ljava.lang.String;@180c26
host=[Ljava.lang.String;@45bb9d
user-agent=[Ljava.lang.String;@204425

"cookie":

JSESSIONID=javax.servlet.http.Cookie@5a142f

"initParam":

Class path list:

/C:/local/jakarta-tomcat-4.1.18/webapps/ROOT/WEB-INF/classes/
/C:/local/jakarta-tomcat-4.1.18/webapps/ROOT/WEB-INF/lib/jstl.jar
/C:/local/jakarta-tomcat-4.1.18/webapps/ROOT/WEB-INF/lib/standard.jar
C:/local/jakarta-tomcat-4.1.18/shared/classes/
C:/local/jakarta-tomcat-4.1.18/common/classes/
C:/local/jakarta-tomcat-4.1.18/common/endorsed/xercesImpl.jar
C:/local/jakarta-tomcat-4.1.18/common/endorsed/xmlParserAPIs.jar
C:/local/jakarta-tomcat-4.1.18/common/lib/activation.jar
C:/local/jakarta-tomcat-4.1.18/common/lib/ant.jar
C:/local/jakarta-tomcat-4.1.18/common/lib/commons-collections.jar
C:/local/jakarta-tomcat-4.1.18/common/lib/commons-dbcp.jar
C:/local/jakarta-tomcat-4.1.18/common/lib/commons-logging-api.jar
C:/local/jakarta-tomcat-4.1.18/common/lib/commons-pool.jar
C:/local/jakarta-tomcat-4.1.18/common/lib/jasper-compiler.jar
C:/local/jakarta-tomcat-4.1.18/common/lib/jasper-runtime.jar
C:/local/jakarta-tomcat-4.1.18/common/lib/jdbc2_0-stdext.jar
C:/local/jakarta-tomcat-4.1.18/common/lib/jndi.jar
C:/local/jakarta-tomcat-4.1.18/common/lib/jta.jar
C:/local/jakarta-tomcat-4.1.18/common/lib/mail.jar
C:/local/jakarta-tomcat-4.1.18/common/lib/naming-common.jar
C:/local/jakarta-tomcat-4.1.18/common/lib/naming-factory.jar
C:/local/jakarta-tomcat-4.1.18/common/lib/naming-resources.jar

C:/local/jakarta-tomcat-4.1.18/common/lib/servlet.jar

JSTL Core Example - JstlPrimeNumbers.jsp

As my second JSTL core example, JstlPrimeNumbers.jsp, is to calculate if a given number is prime number or not.

```
<?xml version="1.0"?>
<jsp:root xmlns:jsp="http://java.sun.com/JSP/Page"
  xmlns:c="http://java.sun.com/jstl/core" version="1.2">
<!-- JstlPrimeNumbers.jsp - It doesn't work, can not change the index
    Copyright (c) 2003 by Dr. Herong Yang
-->
<jsp:directive.page contentType="text/html"/>
<html><body>
<p>Checking prime numbers:</p>
<c:set var="upperLimit" value="${20}"/>
<c:forEach var="i" begin="${3}" end="${upperLimit}">
 <c:set var="isPrime" value="${true}"/>
 <c:forEach var="j" begin="${2}" end="${i-1}">
  <c:if test="${i%j == 0}">
   <c:set var="isPrime" value="${false}"/>
   <!-- We should break the loop here -->
  </c:if>
 </c:forEach>
 <c:choose>
  <c:when test="${isPrime}">
   <c:out value="${i} is a prime number."/><br/>
  </c:when>
  <c:otherwise>
   <c:out value="${i} is a not prime number."/><br/>
  </c:otherwise>
 </c:choose>
</c:forEach>
</body></html>
</jsp:root>
```

I am not showing you the output here. But it is correct. Trust me. As you can see in the source code, I am not breaking the loop on "j" when "i" has already been approved as a non-prime number, because I don't know how.

JSP Custom Tag

What is a Custom Tag?

Custom Tag is an action tag defined by the user through the JSP tag extension facility. It can be used to move JSP page authoring logics and information into a tag Java class, and invoke it by an action tag that is linked to that class. There are two main advantages of using custom tags:

- Repeatable JSP page logics and information can be simplified and centralized into a single tag. For example, we can define a custom tag called <my:copyright/> for producing the copyright information that need to be used on every page of server.
- Moving complex business logics from the JSP to a tag class, so the JSP page author can concentrate on the presentation logics only. For example, we can define a custom tag called <my:userList/> for producing a HTML table filled with a list of users. The tag class will manage how the put each user into a row, and each user property into a column.

I am sure that the functionalities provided by custom tags can also be archived by using JavaBean and scripting elements together. But tags seem to be simpler to use for many unsophisticated JSP page authors.

"Hello world!" Custom Tag

Before we go into any technical details, let me use a very simple example to show you the steps to define and use a custom tag. I want to define a tag called <hy:hello/> to produce the "Hello world!" in the calling JSP page.

1. Writing the tag class. Here is my first tag class, HelloTag.java, which extends the TagSupport class provided in the JSP package:

```
/**
 * HelloTag.java
 * Copyright (c) 2003 by Dr. Herong Yang. All rights reserved.
 */
import java.io.*;
import javax.servlet.jsp.tagext.*;
public class HelloTag extends TagSupport {
  public int doStartTag() {
```

```
      try {
        pageContext.getOut().write("Hello world!");
      } catch (IOException e) {
        System.err.println(e.toString());
      }
      return SKIP_BODY;
    }
  }
```

2. Installing the tag class. I compiled HelloTag.java with JDK 1.3.1, and servlet.jar provided by the Tomcat 4.1.18 server. Like the JavaBean class files, tag class files also need to be installed in the class path of the tomcat server. So I copied the HelloTag.class to \local\jakarta-tomcat-4.1.18\webapps\root\web-inf\classes directory.

3. Writing the tag library descriptor (tld) file. Now, I need to define a tag in a tag library descriptor file to use the tag class. Here is my first tld file, HyTaglib.tld:

```
<?xml version="1.0"?>
<!DOCTYPE taglib PUBLIC
 "-//Sun Microsystems, Inc.//DTD JSP Tag Library 1.2//EN"
 "http://java.sun.com/j2ee/dtds/web-jsptaglibrary_1_2.dtd">
<!-- HyTaglib.tld
    Copyright (c) 2003 by Dr. Herong Yang
-->
<taglib>
<tlib-version>1</tlib-version>
<jsp-version>1.2</jsp-version>
<short-name>Herong's Tag Library</short-name>
<tag>
 <name>hello</name>
 <tag-class>HelloTag</tag-class>
 <body-content>empty</body-content>
</tag>
</taglib>
```

4. Installing the tld file. Tag library descriptor files need to be accessible by the tomcat server. So I copied HyTaglib.tld to \local\jakarta-tomcat-4.1.18\webapps\root\web-inf\tlds directory.

5. Writing the JSP page. To use my first custom tag, I wrote the following JSP page, hello_tag.jsp:

```
<?xml version="1.0"?>
<jsp:root xmlns:jsp="http://java.sun.com/JSP/Page"
   xmlns:hy="urn:jsptld:/WEB-INF/tlds/HyTaglib.tld" version="1.2">
```

```
<!-- hello_tag.jsp
   Copyright (c) 2003 by Dr. Herong Yang
-->
<jsp:directive.page import="HelloTag"/>
<jsp:directive.page contentType="text/html"/>
<html><body>
<hy:hello/>
</body></html>
</jsp:root>
```

6. Viewing the JSP page. To see the output of my JSP page, I copied hello_tag.jsp to \local\jakarta-tomcat-4.1.18\webapps\root, started tomcat 4.1.18 server, and use Internet Explorer (IE) to view http://localhost:8080/hello_tag.jsp.

7. I did get the "Hello world!" message in the IE window. So my hello tag worked perfectly.

8. If you are changing your tag class after it has been loaded by tomcat, you may need to restart tomcat, or click "restart" on the "root" application on the tomcat admin page. "root" application is where I put my JSP pages.

How Custom Tag Works

Here is my understanding of how custom tag works, using the "Hello world!" tag as an example:

- When "hello_tag.jsp" page is requested for the first time, Tomcat server will translate the JSP page into a java class.
- When the <hy:hello/> custom tag is encountered during the translation, Tomcat server will follow the tld file to locate the HelloTag.class file. Note that the tld file is provided in the "jsp:root" element.
- Then Tomcat server will replace the custom tag with some Java code to instantiate an object of the tag class, initialize the object, and call the doStartTag() method.
- To output data into page, you can get an output stream from the pageContext object provided by the JSP tag extension facility, pageContext.getOut(). This is how the "Hello world!" message gets produced in the IE window.
- Note that I have to use "jsp:directive.page" to import HelloTag class into the JSP page, because my HelloTag class has no Java package name and needs to be imported, even if it is located in the class path.

JSP Tag Java Interface

javax.servlet.jsp.tagext.* Package

javax.servlet.jsp.tagext.* is a Java package defined in J2EE (I saw it in J2EE 1.3.1). This package contains:

- **Tag interface** - The base interface required by a tag class to interact with the JSP tag extension facility.
- **IterationTag interface** - Extended from Tag interface to support looping logic within custom tags.
- **BodyTag interface** - Extended from IterationTag interface to allow custom tags to evaluate their own body.
- **TagSupport class** - Dummy implementation of IterationTag interface.
- **BodyTagSupport class** - Dummy implementation of BodyTag interface.

BodyTag Interface

Since BodyTag interface requires deeper knowledge of JSP in order to evaluate the tag body properly, we will discuss it later. For now, let's concentrate on InterationTag interface only. Here is the list of methods required by InterationTag interface and their calling conditions:

- **setPageContext(PageContext pc)** - A help method, called before calling doStartTag() to pass the PageContext object to the tag object.
- **setParent(Tag t)** - A help method, called before calling doStartTag() to pass the parent tag object to the current tag object, in case the current tag is nested within another tag.
- **getParent()** - A help method, called by child tag classes to access grand parent tags.
- **doStartTag()** - An interfacing point, called at the beginning of the process of the tag.
- **doAfterBody()** - An interfacing point, called after processing the body.
- **doEndTag()** - An interfacing point, called when ready to leave the process of the tag.
- **release()** - An interfacing point, called before ending the process of the tag.

The steps used by the JSP tag extension facility to process a custom tag can be summarized as follows:

- Instanciating the tag object.
- Calling setPageContext() and setParent() of the tag object.
- Processing attributes of the tag. See my other notes on how attribute values are passed to the tag object.

- Calling doStartTag() of the tag object, which may return a flag to request for skipping the body to perform a conditional body logic.
- Processing the body of the tag.
- Calling doAfterBody() of the tag object, which may return a flag to request for processing the body again to perform a loop logic on the body.
- Calling doEndTag() of the tag object.
- Calling release() of the tag object.

Implenting BodyTag Interface - TraceTag.java

In order to confirm my understanding of the IterationTag interface, I wrote the following tag class to print a short message from each implemented method to show when it is called. I also used a very simple logic in doAfterTag() to force the tag body being evaluated twice.

```
/**
 * TraceTag.java
 * Copyright (c) 2003 by Dr. Herong Yang. All rights reserved.
 */
package herong;
import java.io.*;
import javax.servlet.jsp.*;
import javax.servlet.jsp.tagext.*;
public class TraceTag implements IterationTag {
  private boolean stop = false;
  private PageContext pc = null;
  private Tag t = null;
  public void setPageContext(PageContext pc) {
    this.pc = pc;
    println("setPageContext() called.");
  }
  public void setParent(Tag t) {
    this.t = t;
    println("setParent() called.");
  }
  public Tag getParent() {
    println("setParent() called.");
    return t;
  }
  public void setMyAtt(String v) {
    println("setMyAtt() called.");
  }
  public int doStartTag() {
    println("doStartTag() called.");
    return EVAL_BODY_INCLUDE;
```

```
  }
  public int doAfterBody() {
    println("doAfterBody() called.");
    if (!stop) {
      stop = true;
      return EVAL_BODY_AGAIN;
    } else {
      return SKIP_BODY;
    }
  }
  public int doEndTag() {
    println("doEndTag() called.");
    return EVAL_PAGE;
  }
  public void release() {
    println("release() called.");
  }
  private void println(String s) {
    try {
      pc.getOut().println(s+"<br/>");
    } catch (IOException e) {
      System.err.println(e.toString());
    }
  }
}
```

In this tag class, I was planning to have one attribute for my trace tag. Attributes of a tag need to be implemented as properties in the tag class. The setMyAtt(String v) method define a property called "myAtt" for my trace tag.

After compiling my trace tag class, I copied TraceTag.class to \local\jakarta-tomcat-4.1.18\webapps\root\web-inf\classes\herong directory. I had to put it under subdirectory "herong", because the tag class was define in "herong" package.

Then I updated my tld file, HyTaglib.tld, with a new tag element:

```
<?xml version="1.0"?>
<!DOCTYPE taglib PUBLIC
 "-//Sun Microsystems, Inc.//DTD JSP Tag Library 1.2//EN"
 "http://java.sun.com/j2ee/dtds/web-jsptaglibrary_1_2.dtd">
<!-- HyTaglib.tld
    Copyright (c) 2003 by Dr. Herong Yang
-->
<taglib>
```

```
<tlib-version>1</tlib-version>
<jsp-version>1.2</jsp-version>
<short-name>Herong's Tag Library</short-name>
<tag>
 <name>trace</name>
 <tag-class>herong.TraceTag</tag-class>
 <body-content>jsp</body-content>
 <attribute>
  <name>myAtt</name>
  <required>false</required>
 </attribute>
</tag>
<!-- other tags -->
</taglib>
```

Since my trace tag will have body, I set body-content to "jsp" instead of "empty". The updated tld file was copied to \local\jakarta-tomcat-4.1.18\webapps\root\web-inf\tlds directory.

To test my trace tag, I wrote the following JSP page, TraceTagTest.jsp:

```
<?xml version="1.0"?>
<jsp:root xmlns:jsp="http://java.sun.com/JSP/Page"
   xmlns:c="http://java.sun.com/jstl/core"
   xmlns:hy="urn:jsptld:/WEB-INF/tlds/HyTaglib.tld" version="1.2">
<!-- TraceTagTest.jsp
    Copyright (c) 2003 by Dr. Herong Yang
-->
<jsp:directive.page contentType="text/html"/>
<html><body>
<hy:trace myAtt="my value">
<jsp:text>JSP body</jsp:text><br/>
</hy:trace>
</body></html>
</jsp:root>
```

Note that I did not use jsp:directive.page to import my trace tag class, because the class in now under "herong" package.

I put the JSP page on my tomcat server, and restarted the server. When requesting this JSP page with IE, I got the following output:

```
setPageContext() called.
setParent() called.
setMyAtt() called.
```

```
doStartTag() called.
JSP body
doAfterBody() called.
JSP body
doAfterBody() called.
doEndTag() called.
```

The output was exactly what I expected. The JSP was indeed evaluated twice as controlled by the returning flag of doAfterBody(). One surpprise is that there was no call to release(). I don't know why.

The Servlet Class - TraceTagTest_jsp.java

As we all know that JSP pages are translated into Servlet classes before execution. So we actually review the translate Servlet class to see how the tag extension facility connects the Servlet class to the tag class.

To review the Servlet class translated from TraceTagTest.jsp, open TraceTagTest_jsp.java in \local\jakarta-tomcat-4.1.18\work\standalone\localhost_, you will see:

```
package org.apache.jsp;

import javax.servlet.*;
import javax.servlet.http.*;
import javax.servlet.jsp.*;
import org.apache.jasper.runtime.*;

public class TraceTagTest_jsp extends HttpJspBase {

  private static java.util.Vector _jspx_includes;

  private org.apache.jasper.runtime.TagHandlerPool
    _jspx_tagPool_hy_trace_myAtt;

  public TraceTagTest_jsp() {
   _jspx_tagPool_hy_trace_myAtt
      = new org.apache.jasper.runtime.TagHandlerPool();
  }

  public java.util.List getIncludes() {
   return _jspx_includes;
  }

  public void _jspDestroy() {
```

```
  _jspx_tagPool_hy_trace_myAtt.release();
}

public void _jspService(HttpServletRequest request,
   HttpServletResponse response)
      throws java.io.IOException, ServletException {

  JspFactory _jspxFactory = null;
  javax.servlet.jsp.PageContext pageContext = null;
  HttpSession session = null;
  ServletContext application = null;
  ServletConfig config = null;
  JspWriter out = null;
  Object page = this;
  JspWriter _jspx_out = null;

  try {
    _jspxFactory = JspFactory.getDefaultFactory();
    response.setContentType("text/html;charset=UTF-8");
    pageContext = _jspxFactory.getPageContext(this, request, response,
                  null, true, 8192, true);
    application = pageContext.getServletContext();
    config = pageContext.getServletConfig();
    session = pageContext.getSession();
    out = pageContext.getOut();
    _jspx_out = out;

    out.write("<html>");
    out.write("<body>");
    if (_jspx_meth_hy_trace_0(pageContext))
      return;
    out.write("</body>");
    out.write("</html>");
  } catch (Throwable t) {
    out = _jspx_out;
    if (out != null && out.getBufferSize() != 0)
      out.clearBuffer();
    if (pageContext != null) pageContext.handlePageException(t);
  } finally {
    if (_jspxFactory != null)
       _jspxFactory.releasePageContext(pageContext);
  }
}
```

```
  private boolean _jspx_meth_hy_trace_0(
    javax.servlet.jsp.PageContext pageContext)
        throws Throwable {
   JspWriter out = pageContext.getOut();
   /* ----  hy:trace ---- */
   herong.TraceTag _jspx_th_hy_trace_0 = (herong.TraceTag)
     _jspx_tagPool_hy_trace_myAtt.get(herong.TraceTag.class);
   _jspx_th_hy_trace_0.setPageContext(pageContext);
   _jspx_th_hy_trace_0.setParent(null);
   _jspx_th_hy_trace_0.setMyAtt("my value");
   int _jspx_eval_hy_trace_0 = _jspx_th_hy_trace_0.doStartTag();
   if (_jspx_eval_hy_trace_0 !=
     javax.servlet.jsp.tagext.Tag.SKIP_BODY) {
    do {
      out.write("JSP body");
      out.write("<br/>");
      int evalDoAfterBody = _jspx_th_hy_trace_0.doAfterBody();
      if (evalDoAfterBody !=
        javax.servlet.jsp.tagext.BodyTag.EVAL_BODY_AGAIN)
       break;
    } while (true);
   }
   if (_jspx_th_hy_trace_0.doEndTag() ==
     javax.servlet.jsp.tagext.Tag.SKIP_PAGE)
    return true;
   _jspx_tagPool_hy_trace_myAtt.reuse(_jspx_th_hy_trace_0);
   return false;
  }
}
```

As you can see, the entire hy:trace element was translated into a method call, _jspx_meth_hy_trace_0(pageContext). In that method, an object was instantiated from my trace tag class. Then interface methods were called one by one in the same order as we discussed in the previous section. The "do" loop was there to re-evaluate the body based on the returning flag of doAfterBody() call.

After finishing this trace tag example, I had a very good understanding of how the tag interface offered by the extension facility works now. How about you?

Dummy Implementation of IterationTag Interface - TagSupport Class

J2EE supplied a dummy implementation, called TagSupport, to the IterationTag interface. If you are design a new tag class, you may extend your class from TagSupport, instead of

implement InterationTag. Extending your tag class from TagSupport gives you two advantages.

1. Need to code any required method, if you are happy with the implementation of that method inside TagSupport.

2. TagSupport provides you the pageContext object ready to use. No need for you to code setPageContext() and save it in your tag object.

Examples of using TagSupport will be include in my other notes on custom tags.

JSP Tag Attribute Handling

Tag Attribute Setter Method

The JSP tag extension facility maps each attribute encountered in a custom tag to an property of the same name of the tag object. So if you want to use an attribute in a custom tag, you must define a property in the tag class with the same name as the attribute. Here is the steps you need to follow to add an attribute to a custom tag:

1. Define a property in the tag class with the same name as the attribute name by adding a setter method and a getter method. This is the same way as defining a property in a JavaBean class.

2. Add an "attribute" element inside the "tag" element in your TLD file with the following syntax:

```
<tag>
 <name>tag_name</name>
 <tag-class>class_full_name</tag-class>
 <body-content>empty | jsp</body-content>
 <attribute>
  <name>attribute_name</name>
  <required>true | false</required>
 </attribute>
</tag>
```

3. Add the attribute to the custom tag in your JSP page. Then the tag is processed, the setter method of the tag object will be called to pass the attribute value into the tag object.

Tag Attribute Setter Method Example - EchoTag.java

To show you how to use attributes in a custom tag, I wrote the following example tag, EchoTag.java. It does nothing but takes the value of the "message" attribute, and echoes back to the page output with characters reversed.

```
/**
 * EchoTag.java
 * Copyright (c) 2003 by Dr. Herong Yang. All rights reserved.
 */
```

```
package herong;
import java.io.*;
import javax.servlet.jsp.tagext.*;
public class EchoTag extends TagSupport {
  private String message = null;
  public void setMessage(String m) {
    message = m;
  }
  public int doStartTag() {
    try {
      if (message!=null) {
        char[] a = message.toCharArray();
        int n = a.length;
        for (int i=0; i<n/2; i++) {
          char t = a[i];
          a[i] = a[n-1-i];
          a[n-i-1] = t;
        }
        pageContext.getOut().print(a);
        pageContext.getOut().println("<br/>");
      }
    } catch (IOException e) {
      System.err.println(e.toString());
    }
    return SKIP_BODY;
  }
}
```

Here is the TLD file:

```
<?xml version="1.0"?>
<!DOCTYPE taglib PUBLIC
 "-//Sun Microsystems, Inc.//DTD JSP Tag Library 1.2//EN"
 "http://java.sun.com/j2ee/dtds/web-jsptaglibrary_1_2.dtd">
<!-- HyTaglib.tld
    Copyright (c) 2003 by Dr. Herong Yang
-->
<taglib>
<tlib-version>1</tlib-version>
<jsp-version>1.2</jsp-version>
<short-name>Herong's Tag Library</short-name>
<tag>
 <name>echo</name>
 <tag-class>herong.EchoTag</tag-class>
```

```
 <body-content>empty</body-content>
 <attribute>
  <name>message</name>
  <required>false</required>
 </attribute>
</tag>
</taglib>
```

Here is a test page, EchoTagTest.jsp:

```
<?xml version="1.0"?>
<jsp:root xmlns:jsp="http://java.sun.com/JSP/Page"
   xmlns:c="http://java.sun.com/jstl/core"
   xmlns:hy="urn:jsptld:/WEB-INF/tlds/HyTaglib.tld" version="1.2">
<!-- EchoTagTest.jsp
    Copyright (c) 2003 by Dr. Herong Yang
-->
<jsp:directive.page contentType="text/html"/>
<html><body>
<hy:echo message="Fish, I love you and respect you very much."/>
</body></html>
</jsp:root>
```

You can guess what you will be getting when you access this page.

Tag Attribute Value Type Conversion

In my previous example, the attribute value is passed to the tag object property as a string data type. Can we pass attribute values to tag object as other data types or objects? The answer is yes. JSP tag extension facility will try to convert attribute values to the data types of the tag object properties. I couldn't find any official specifications on the match rules. The following is my guesses:

- Match the attribute name to the setter method of a property with the same name in the tag object.
- If the data type of the input parameter of the setter method is a primitive data type, like "long", give the attribute value as literal data to the input parameter.
- If the data type of the input parameter of the setter method is the wrapper class of a primitive data type, like "Long", instantiate an object of that class with the attribute value as a String input to the constructor, and give the new object to the input parameter.
- If the data type of the input parameter of the setter method is the String class, give the attribute value as string literal to the input parameter.
- If the data type of the input parameter of the setter method is any other class, unknown behavior.

Tag Attribute Value Type Conversion Example - AttValueTag.java

Here is an example of to show you how JSP tag extension facility is converting the attribute values to the tag's setter methods required data types.

The tag class:

```
/**
 * AttValueTag.java
 * Copyright (c) 2003 by Dr. Herong Yang. All rights reserved.
 */
package herong;
import java.io.*;
import javax.servlet.jsp.*;
import javax.servlet.jsp.tagext.*;
public class AttValueTag extends TagSupport {
  private long longValue = 0;
  private Long longObject = null;
  private double doubleValue = 0.0;
  private Double doubleObject = null;
  private boolean booleanValue = true;
  private Boolean booleanObject = null;
  private String stringObject = null;
  public void setLongValue(long val) {
    longValue = val;
  }
  public void setLongObject(Long obj) {
    longObject = obj;
  }
  public void setDoubleValue(double val) {
    doubleValue = val;
  }
  public void setDoubleObject(Double obj) {
    doubleObject = obj;
  }
  public void setBooleanValue(boolean val) {
    booleanValue = val;
  }
  public void setBooleanObject(Boolean obj) {
    booleanObject = obj;
  }
  public void setStringObject(String obj) {
    stringObject = obj;
```

```
  }
  public int doStartTag() {
    JspWriter out = pageContext.getOut();
    try {
      out.println("longValue = "+longValue+"<br/>");
      out.println("longObject = "+longObject+"<br/>");
      out.println("doubleValue = "+doubleValue+"<br/>");
      out.println("doubleObject = "+doubleObject+"<br/>");
      out.println("booleanValue = "+booleanValue+"<br/>");
      out.println("booleanObject = "+booleanObject+"<br/>");
      out.println("stringObject = "+stringObject+"<br/>");
    } catch (IOException e) {
      System.err.println(e.toString());
    }
    return SKIP_BODY;
  }
}
```

The TLD file:

```
<?xml version="1.0"?>
<!DOCTYPE taglib PUBLIC
 "-//Sun Microsystems, Inc.//DTD JSP Tag Library 1.2//EN"
 "http://java.sun.com/j2ee/dtds/web-jsptaglibrary_1_2.dtd">
<!-- HyTaglib.tld
    Copyright (c) 2003 by Dr. Herong Yang
-->
<taglib>
<tlib-version>1</tlib-version>
<jsp-version>1.2</jsp-version>
<short-name>Herong's Tag Library</short-name>
<tag>
 <name>attValue</name>
 <tag-class>herong.AttValueTag</tag-class>
 <body-content>empty</body-content>
 <attribute>
  <name>longValue</name>
  <required>false</required>
 </attribute>
 <attribute>
  <name>longObject</name>
  <required>false</required>
 </attribute>
 <attribute>
```

```
  <name>doubleValue</name>
  <required>false</required>
 </attribute>
 <attribute>
  <name>doubleObject</name>
  <required>false</required>
 </attribute>
 <attribute>
  <name>booleanValue</name>
  <required>false</required>
 </attribute>
 <attribute>
  <name>booleanObject</name>
  <required>false</required>
 </attribute>
 <attribute>
  <name>stringObject</name>
  <required>false</required>
 </attribute>
</tag>

</taglib>
```

The JSP file:

```
<?xml version="1.0"?>
<jsp:root xmlns:jsp="http://java.sun.com/JSP/Page"
   xmlns:c="http://java.sun.com/jstl/core"
   xmlns:hy="urn:jsptld:/WEB-INF/tlds/HyTaglib.tld" version="1.2">
<!-- AttValueTagTest.jsp
    Copyright (c) 2003 by Dr. Herong Yang
-->
<jsp:directive.page contentType="text/html"/>
<html><body>
<hy:attValue longObject="1" longValue="2"
 doubleObject="1.1" doubleValue="2.2"
 booleanObject="true" booleanValue="false"
 stringObject="Hello world!"/>
</body></html>
</jsp:root>
```

The output:

```
longValue = 2
```

```
longObject = 1
doubleValue = 2.2
doubleObject = 1.1
booleanValue = false
booleanObject = true
stringObject = Hello world
```

The Servlet class produced by Tomcat server:

```
...
  private boolean _jspx_meth_hy_attValue_0(
     javax.servlet.jsp.PageContext pageContext)
         throws Throwable {
    JspWriter out = pageContext.getOut();
    /* ----  hy:attValue ---- */
    herong.AttValueTag _jspx_th_hy_attValue_0 = (herong.AttValueTag)
    _jspx_tagPool_hy_attValue_stringObject_longValue_longObject...
    _jspx_th_hy_attValue_0.setPageContext(pageContext);
    _jspx_th_hy_attValue_0.setParent(null);
    _jspx_th_hy_attValue_0.setLongObject(new Long(1l));
    _jspx_th_hy_attValue_0.setLongValue(2l);
    _jspx_th_hy_attValue_0.setDoubleObject(new Double(1.1));
    _jspx_th_hy_attValue_0.setDoubleValue(2.2);
    _jspx_th_hy_attValue_0.setBooleanObject(new Boolean(true));
    _jspx_th_hy_attValue_0.setBooleanValue(false);
    _jspx_th_hy_attValue_0.setStringObject("Hello world!");
    int _jspx_eval_hy_attValue_0 = _jspx_th_hy_attValue_0.doStartTag();
    if (_jspx_th_hy_attValue_0.doEndTag() ==
       javax.servlet.jsp.tagext.Tag.SKIP_PAGE)
      return true;
    _jspx_tagPool_hy_attValue_stringObject_longValue_longObject...
    return false;
  }
...
```

I only copied the method that interacts with the tag class out of the Servlet class here. Note that:

- The setter methods for the primitive data types are called with attribute values directly, except that "l" is added to the end of the value, like "1l".
- The setter methods for the wrapper classes of the primitive data types are called with new objects instantiated with the attribute values.
- How can Tomcat server figure out the class type of the setter method parameter? Could someone help me here?

Tag Attribute Value Expression

In the previous sections, I only talked about how to define an attribute in a custom tag, how attribute values are passed to the tag class, and how attribute values are converted to correct data type required by the tag class. Now let's look at the possibility of entering expressions as attribute values in a custom tag, similar to JSTL tags.

Based on my readings on the Internet, expression language is not supported directly in custom tags in JSP 1.2. However there are two approaches to use expressions indirectly in custom tags.

1. Using Java expression elements as attribute values. You need to define the attribute in the TLD file with <rtexprvalue>true</rtexprvalue>, then enter the attribute value with the following format:

```
<hy:tag att="<%=java_expression%>"/>
```

2. Using JSTL expressions and evaluate them inside tag class. You can enter a JSTL expression in an attribute value, received it by the setter method as a string, then evaluate it to the desire data type using the expression evaluation tool offer by the JSTL.

The first approach requires the JSP page to be written in a non-XML format. And I don't like that format. So I am not going to try that.

The second approach seems to be interesting and powerful. You can use it to pass an object of any class as an attribute value to the tag class. See the example bellow.

Tag Attribute Value Expression Example - AttObjectTag.java

The tag class:

```
/**
 * AttObjectTag.java
 * Copyright (c) 2003 by Dr. Herong Yang. All rights reserved.
 */
package herong;
import java.util.*;
import java.io.*;
import javax.servlet.jsp.*;
import javax.servlet.jsp.tagext.*;
import org.apache.taglibs.standard.lang.support.*;
public class AttObjectTag extends TagSupport {
   private String booleanAtt = null;
   private Boolean booleanObject = null;
```

```
private String stringAtt = null;
private String stringObject = null;
private String mapAtt = null;
private Map mapObject = null;
public void setBooleanAtt(String att) {
  booleanAtt = att;
  try {
    booleanObject =
      (Boolean) ExpressionEvaluatorManager.evaluate(
      "booleanAtt", booleanAtt, java.lang.Boolean.class,
      this, pageContext);
  } catch (JspException e) {
    System.err.println(e.toString());
  }
}
public void setStringAtt(String att) {
  stringAtt = att;
  try {
    stringObject =
      (String) ExpressionEvaluatorManager.evaluate(
      "stringAtt", stringAtt, java.lang.String.class,
      this, pageContext);
  } catch (JspException e) {
    System.err.println(e.toString());
  }
}
public void setMapAtt(String att) {
  mapAtt = att;
  try {
    mapObject =
      (Map) ExpressionEvaluatorManager.evaluate(
      "mapAtt", mapAtt, java.util.Map.class,
      this, pageContext);
  } catch (JspException e) {
    System.err.println(e.toString());
  }
}
public int doStartTag() {
  JspWriter out = pageContext.getOut();
  try {
    out.println("booleanAtt = "+booleanAtt+"<br/>");
    out.println("booleanObject = "+booleanObject+"<br/>");
    out.println("stringAtt = "+stringAtt+"<br/>");
    out.println("stringObject = "+stringObject+"<br/>");
```

```
        out.println("mapAtt = "+mapAtt+"<br/>");
        if (mapObject!=null) {
          out.println("mapObject.size = "+mapObject.size()+"<br/>");
        } else {
          out.println("mapObject = null<br/>");
        }
      } catch (IOException e) {
        System.err.println(e.toString());
      }
      return SKIP_BODY;
    }
}
```

The TLD file:

```
<?xml version="1.0"?>
<!DOCTYPE taglib PUBLIC
 "-//Sun Microsystems, Inc.//DTD JSP Tag Library 1.2//EN"
 "http://java.sun.com/j2ee/dtds/web-jsptaglibrary_1_2.dtd">
<!-- HyTaglib.tld
    Copyright (c) 2003 by Dr. Herong Yang
-->
<taglib>
<tlib-version>1</tlib-version>
<jsp-version>1.2</jsp-version>
<short-name>Herong's Tag Library</short-name>
<tag>
 <name>attObject</name>
 <tag-class>herong.AttObjectTag</tag-class>
 <body-content>empty</body-content>
 <attribute>
  <name>booleanAtt</name>
  <required>false</required>
 </attribute>
 <attribute>
  <name>stringAtt</name>
  <required>false</required>
 </attribute>
 <attribute>
  <name>mapAtt</name>
  <required>false</required>
 </attribute>
</tag>
```

```
</taglib>
```

The JSP page:

```
<?xml version="1.0"?>
<jsp:root xmlns:jsp="http://java.sun.com/JSP/Page"
  xmlns:c="http://java.sun.com/jstl/core"
  xmlns:hy="urn:jsptld:/WEB-INF/tlds/HyTaglib.tld" version="1.2">
<!-- AttObjectTagTest.jsp
    Copyright (c) 2003 by Dr. Herong Yang
-->
<jsp:directive.page contentType="text/html"/>
<html><body>
<p>Regular Strings:</p>
<hy:attObject booleanAtt="true" stringAtt="Hello world!"/>
<p>Literals:</p>
<hy:attObject booleanAtt="${false}" stringAtt="${'Herong Yang'}"/>
<p>Expressions:</p>
<hy:attObject booleanAtt="${1==1}"
  stringAtt="${pageContext.request.method}"
  mapAtt="${cookie}"/>
</body></html>
</jsp:root>
```

The output:

```
Regular Strings:

booleanAtt = true
booleanObject = true
stringAtt = Hello world!
stringObject = Hello world!
mapAtt = null
mapObject = null

Literals:

booleanAtt = ${false}
booleanObject = false
stringAtt = ${'Herong Yang'}
stringObject = Herong Yang
mapAtt = null
mapObject = null
```

Expressions:

booleanAtt = ${1==1}
booleanObject = true
stringAtt = ${pageContext.request.method}
stringObject = GET
mapAtt = ${cookie}
mapObject.size = 0

Note that:

- The JSTL expression evaluation tool is a static method, evaluate(), in ExpressionEvaluationManager class, in org.apache.taglibs.standard.lang.support package.
- The expression rules are identical to those described in JSTL, because we are using the same evaluation tool.
- If you click "refresh" button on the browser, the mapObject.size will be 1, because the session id is stored in the cookie map.

Tags Working Together

Nested Tags

When tags are nested, JSP tag extension facility offers the child tag an access to the parent tag by the setParent() method call. You can take advantage of this access to design nested tags to work together to perform more complex functions.

As an example, I wrote two simple tags to perform a loop function with the ability to break out the loop conditionally.

Here is my loop tag, LoopTag.java:

```
/**
 * LoopTag.java
 * Copyright (c) 2003 by Dr. Herong Yang. All rights reserved.
 */
package herong;
import java.io.*;
import javax.servlet.jsp.tagext.*;
public class LoopTag extends TagSupport {
   public boolean stop = false;
   public int doStartTag() {
      return EVAL_BODY_INCLUDE;
   }
   public int doAfterBody() {
      if (!stop) return EVAL_BODY_AGAIN;
      else return SKIP_BODY;
   }
}
```

The logic of the "loop" tag is very simple. It evaluates its body immediately once the tag is encountered. At the end of the evaluation, it checks its "stop" condition, if "stop" is false, it evaluates its body again. This loop will be stopped only if "stop" is set to true, during the body evaluation.

Here is the break tag, BreakTag.java:

```
/**
 * BreakTag.java
 * Copyright (c) 2003 by Dr. Herong Yang. All rights reserved.
 */
package herong;
import herong.*;
import javax.servlet.jsp.tagext.*;
public class BreakTag extends TagSupport {
   private Tag parent = null;
```

```
   public void setParent(Tag t) {
      parent = t;
   }
   public int doStartTag() {
      if (parent!=null) {
         LoopTag grandParent = (LoopTag) parent.getParent();
         if (grandParent!=null) grandParent.stop = true;
      }
      return SKIP_BODY;
   }
}
```

The logic of the "break" tag is a little bit more complex. To break the "loop" tag conditionally, we are expecting an "if" tag will be nested inside the "loop" tag to control the condition, and the "break" tag will be nested inside the "if" tag to break the loop. So the "break" tag is actually nested 2 levels down inside the "loop" tag. This is why I have to use "parent.getParent()" to get the grand parent to access the "loop" tag.

Here is the TLD file:

```
<?xml version="1.0"?>
<!DOCTYPE taglib PUBLIC
 "-//Sun Microsystems, Inc.//DTD JSP Tag Library 1.2//EN"
 "http://java.sun.com/j2ee/dtds/web-jsptaglibrary_1_2.dtd">
<!-- HyTaglib.tld
    Copyright (c) 2003 by Dr. Herong Yang
-->
<taglib>
<tlib-version>1</tlib-version>
<jsp-version>1.2</jsp-version>
<short-name>Herong's Tag Library</short-name>
<tag>
 <name>loop</name>
 <tag-class>herong.LoopTag</tag-class>
 <body-content>jsp</body-content>
</tag>
<tag>
 <name>break</name>
 <tag-class>herong.BreakTag</tag-class>
 <body-content>empty</body-content>
</tag>
<!-- other tags -->
</taglib>
```

Here is a test page, LoopTagTest.jsp:

```
<?xml version="1.0"?>
```

```
<jsp:root xmlns:jsp="http://java.sun.com/JSP/Page"
  xmlns:c="http://java.sun.com/jstl/core"
  xmlns:hy="urn:jsptld:/WEB-INF/tlds/HyTaglib.tld" version="1.2">
<!-- LoopTagTest.jsp
   Copyright (c) 2003 by Dr. Herong Yang
-->
<jsp:directive.page contentType="text/html"/>
<html><body>
<c:set var="i" value="${1}"/>
<c:set var="s" value="${0}"/>
<hy:loop>
 <c:set var="s" value="${s+i}"/>
 <c:set var="i" value="${i+1}"/>
 <c:if test="${i gt 10}">
  <hy:break/>
 </c:if>
</hy:loop>
<c:out value="Sum = ${s}"/><br/>
</body></html>
</jsp:root>
```

I am sure you can read this page, and understand what I am doing with the loop. Warning, my "break" tag is not a truly break statement. If there is any additional body content in the loop tag after the "break" tag, it will not be skipped.

Sharing Data with Other Tags

In the previous example, we looked at how tags can be nested inside each other, and how child tags can access parent tags to modify their behavior. In the next example, we will look at how non-nested tags, brother tags, can communicate to each other.

If a tag wants to share data with a brother tag, it must store the data to a common place where both of them have access. Obviously, that common place is the pageContext object. The JSP tag extension facility offers to every tag object the access to pageContext object by the setPageContext() call in Tag interface. If you use the TagSupport implementation class, pageContext is already made available as an instance variable.

If you read the PageContext class API, you will see that it allows you to store and retrieve objects as named attributes at any time. So if one tag wants to share an object to another tag, it can store that object to pageContext; and the other tag can retrieve it from pageContext.

Another advantage of using pageContext to share objects is that JSTL tags are also using pageContext to store and share objects. So if we use it correctly, we can share objects in custom tags with JSTL tags.

Let's look at a very simple example, SetTimeTag.java. It does nothing but storing the current time in milliseconds into pageContext as an attribute with a given name. Once the data is stored in pageContext, any other tags can come and retrieve it.

```
/**
 * SetTimeTag.java
 * Copyright (c) 2003 by Dr. Herong Yang. All rights reserved.
 */
package herong;
import javax.servlet.jsp.tagext.*;
public class SetTimeTag extends TagSupport {
  private String var = null;
  public void setVar(String v) {
    this.var = v;
  }
  public int doStartTag() {
    Long now = new Long(System.currentTimeMillis());
    pageContext.setAttribute(var,now);
    return SKIP_BODY;
  }
}
```

The TLD file:

```
<?xml version="1.0"?>
<!DOCTYPE taglib PUBLIC
 "-//Sun Microsystems, Inc.//DTD JSP Tag Library 1.2//EN"
 "http://java.sun.com/j2ee/dtds/web-jsptaglibrary_1_2.dtd">
<!-- HyTaglib.tld
    Copyright (c) 2003 by Dr. Herong Yang
-->
<taglib>
<tlib-version>1</tlib-version>
<jsp-version>1.2</jsp-version>
<short-name>Herong's Tag Library</short-name>
<tag>
 <name>setTime</name>
 <tag-class>herong.SetTimeTag</tag-class>
 <body-content>empty</body-content>
 <attribute>
  <name>var</name>
  <required>true</required>
 </attribute>
```

```
</tag>
<!-- other tags -->
</taglib>
```

Here is the testing page:

```
<?xml version="1.0"?>
<jsp:root xmlns:jsp="http://java.sun.com/JSP/Page"
  xmlns:c="http://java.sun.com/jstl/core"
  xmlns:hy="urn:jsptld:/WEB-INF/tlds/HyTaglib.tld" version="1.2">
<!-- SetTimeTagTest.jsp
    Copyright (c) 2003 by Dr. Herong Yang
-->
<jsp:directive.page contentType="text/html"/>
<html><body>
<hy:setTime var="t1"/>
<p>Checking prime numbers:</p>
<c:set var="upperLimit" value="${50}"/>
<c:forEach var="i" begin="${3}" end="${upperLimit}">
 <c:set var="isPrime" value="${true}"/>
 <c:forEach var="j" begin="${2}" end="${i-1}">
  <c:if test="${i%j == 0}">
   <c:set var="isPrime" value="${false}"/>
   <!-- We should break the loop here -->
  </c:if>
 </c:forEach>
 <c:choose>
  <c:when test="${isPrime}">
   <c:out value="${i} is a prime number."/><br/>
  </c:when>
  <c:otherwise>
   <c:out value="${i} is a not prime number."/><br/>
  </c:otherwise>
 </c:choose>
</c:forEach>
<hy:setTime var="t2"/>
<p><c:out value="Total time = ${t2-t1} milliseconds."/></p>
</body></html>
</jsp:root>
```

The output:

```
Checking prime numbers:
```

3 is a prime number.
4 is a not prime number.
5 is a prime number.
6 is a not prime number.
7 is a prime number.
8 is a not prime number.
9 is a not prime number.
10 is a not prime number.
...
50 is a not prime number.

Total time = 110 milliseconds.

As you can see from the testing page, I used "setTime" to store the current time at the beginning of my prime number checking process as "t1", and the current time at the end as "t2". Then I used "out" to retrieve them and calculate their difference in a single expression.

Tomcat 4.1.18 with JDK 1.4.1

This chapter explains:

- How to change Tomcat 4.1.18 installation from using JDK 1.3.1 to using JDK 1.4.1.
- Compilation issues of using JavaBean classes in unnamed packages.
- How to create JavaBean classes in named packages.

Upgrading Tomcat 4.1.18 to JDK 1.4.1

One of the readers reported a JSP compilation issue while following my tutorials with Tomcat 4.1.29 and JDK 1.4.x. In order to understand the issue better, I upgraded my Tomcat 4.1.18 installation from JDK 1.3.1 to JDK 1.4.1:

1. Checked my JDK 1.4.1 installation. The following command shows that I have JDK 1.4.1_01 working correctly:

```
>\local\j2sdk1.4.1\bin\java -version
java version "1.4.1_01"
Java(TM) 2 Runtime Environment, Standard Edition (build 1.4.1_01-b01)
Java HotSpot(TM) Client VM (build 1.4.1_01-b01, mixed mode)
```

2. Started Tomcat server with JDK 1.4.1 without any problem:

```
cd \local\jakarta-tomcat-4.1.18\bin
set JAVA_HOME=\local\j2sdk1.4.1
startup
```

3. Created the following JSP page, hello.jsp, and copied to \local\jakarta-tomcat-4.1.18\webapps\ROOT:

```
<html><body>
<% out.println("Hello world!"); %>
</body></html>
```

4. Ran Internet Explorer (IE) with url: http://localhost:8080/hello.jsp. You should see "Hello world!" in the IE window.

I am pretty sure now my Tomcat server is running ok with JDK 1.4.1.

Compilation Errors with Tomcat 4.1.18 and JDK 1.4.1

To demonstrate the compilation error, run IE on the following JSP page I created in the "Using JavaBean Classes" chapter, UseBean.jsp

```
<?xml version="1.0"?>
<jsp:root xmlns:jsp="http://java.sun.com/JSP/Page" version="1.2">
<!-- UseBean.jsp
    Copyright (c) 2002 by Dr. Herong Yang
-->
<html><body>
<jsp:directive.page import="CacheBean"/>
<jsp:useBean id="b" class="CacheBean"/>
<jsp:setProperty name="b" property="text" value="Hello world!"/>
Property from my Bean:
<jsp:getProperty name="b" property="text"/>
<br/>
Info from my Bean:
<jsp:expression>b.getInfo()</jsp:expression>
</body></html>
</jsp:root>
```

You will see a compilation error in the IE window:

```
org.apache.jasper.JasperException: Unable to compile class for JSP

An error occurred at line: 8 in the jsp file: /UseBean.jsp

Generated servlet error:
   [javac] Compiling 1 source file

C:\local\jakarta-tomcat-4.1.18\work\Standalone\localhost
  \_\UseBean_jsp.java:7: '.' expected
import CacheBean;
```

This tells us that JDK 1.4.1 does not allow import statement to be used on classes in unnamed packages. So what can we do about this? Can we remove the import statement? The answer is no. You can try by running the following modified JSP page, UseBeanModified.jsp:

```
<?xml version="1.0"?>
<jsp:root xmlns:jsp="http://java.sun.com/JSP/Page" version="1.2">
```

```
<!-- UseBeanModified.jsp
    Copyright (c) 2002 by Dr. Herong Yang
-->
<html><body>
<jsp:useBean id="b" class="CacheBean"/>
<jsp:setProperty name="b" property="text" value="Hello world!"/>
Property from my Bean:
<jsp:getProperty name="b" property="text"/>
<br/>
Info from my Bean:
<jsp:expression>b.getInfo()</jsp:expression>
</body></html>
</jsp:root>
```

You will get the following compilation error saying CacheBean is not resolvable:

```
org.apache.jasper.JasperException: Unable to compile class for JSP

An error occurred at line: 7 in the jsp file: /jsp/UseBeanModified.jsp

Generated servlet error:
    [javac] Compiling 1 source file

D:\local\jakarta-tomcat-4.1.18\work\Standalone\localhost
  \_\jsp\UseBeanModified_jsp.java:43: cannot resolve symbol
symbol  : class CacheBean
location: class org.apache.jsp.UseBeanModified_jsp
      CacheBean b = null;
      ^
```

The actual cause of this error is not the compiler. It is the Tomcat compilation environment. Because you can compile the converted class, UseBeanModified_jsp.java, correctly in a command window with the same class paths used by the Tomcat environment.

The only option left to use a JavaBean with Tomcat 4.1.18 and JDK 1.4.1 is to create your JavaBean in a named package. The next section will give you an example.

JavaBean in a Named Package - TempratureConvertorBean.java

Creating JavaBean classes in named packages is very simple, just add a package declaration statement at the beginning of the class. The following example shows you a JavaBean classe defined in a package named as "herong":

```
/**
 * TempratureConvertorBean.java
 * Copyright (c) 2003 by Dr. Herong Yang. All rights reserved.
 */
package herong;
public class TempratureConvertorBean {
  private double celsius = 0.0;
  private double fahrenheit = 32.0;
  public double getCelsius() {
    return celsius;
  }
  public void setCelsius(double c) {
    celsius = c;
    fahrenheit = 1.8*c + 32.0;
  }
  public double getFahrenheit() {
    return fahrenheit;
  }
  public void setFahrenheit(double f) {
    fahrenheit = f;
    celsius = (f-32.0)/1.8;
  }
  public String getInfo() {
    return new String("My TempraturConvertorBean - Version 1.00");
  }
}
```

Compile this source code, and copy the class file to the Tomcat class path. Remember to store the class file under a sub directory named as "herong".

```
>\local\j2sdk1.4.1\bin\java TempratureConvertorBean.java
>copy TempratureConvertorBean.class
   \local\jakarta-tomcat-4.1.18\webapps\root\web-inf\classes\herong
```

Now we are ready to test this JavaBean with an JSP page, TempratureConvertor.jsp:

```
<?xml version="1.0"?>
<jsp:root xmlns:jsp="http://java.sun.com/JSP/Page" version="1.2">
<!-- TempratureConvertor.jsp
    Copyright (c) 2003 by Dr. Herong Yang
-->
<jsp:directive.page contentType="text/html"/>
<html><body>
```

```
<jsp:useBean id="b" class="herong.TempratureConvertorBean"/>
<jsp:expression>b.getInfo()</jsp:expression><br/>
Setting property "fahrenheit" to 70.0.<br/>
<jsp:setProperty name="b" property="fahrenheit" value="70.0"/>
Getting property "celsius" back:
<jsp:getProperty name="b" property="celsius"/>
<br/>
</body></html>
</jsp:root>
```

Open this JSP page with IE, you will get:

```
My TempraturConvertorBean - Version 1.00
Setting property "fahrenheit" to 70.0.
Getting property "celsius" back: 21.11111111111111
```

It works! Note that there is no need to use the import statement, if you use the fully qualified class name in the jsp:useBean action element.

Conclusions:

- Using JavaBean classes in unnamed packages with Tomcat 4.1.18 and JDK 1.3.1 requires import directive elements in the JSP pages.
- There is no way to use JavaBean classes in unnamed packages with Tomcat 4.1.18 and JDK 1.4.1.
- There is no problem using JavaBean classes in named packages with Tomcat 4.1.18 and JDK 1.4.1.

References

- *JavaServer Pages*, Sun Mirosystems, http://java.sun.com/products/jsp/
- *JSR-000152 JavaServer PagesTM 2.0 Specification*, Sun Mirosystems, http://jcp.org/aboutJava/communityprocess/first/jsr152/index2.html
- *JSR-000053 JavaTM Servlet 2.3 and JavaServer PagesTM 1.2 Specifications*, Java Community Process, http://www.jcp.org/aboutJava/communityprocess/first/jsr053/
- *Apache Jakarta Project*, Apache Software Foundation, http://jakarta.apache.org/tomcat/index.html
- *Taglibs*, Apache Software Foundation, http://jakarta.apache.org/taglibs/index.html
- *Servlets and JavaServer Pages (JSP) 1.0: A Tutorial*, Marty Hall, 1999, http://www.apl.jhu.edu/~hall/java/Servlet-Tutorial/
- *JSR-000052 JavaServer PagesTM Standard Tag Library Specification*, Java Community Process, http://jcp.org/aboutJava/communityprocess/final/jsr052/
- *JavaServer Pages Standard Tag Library*, Stephanie Bodoff, http://java.sun.com/webservices/docs/1.0/tutorial/doc/JSTL.html#74644